

Structuring an Abstract Interpreter through Value and State Abstractions: EVA, an Evolved Value Analysis for Frama-C

David Bühler

► To cite this version:

David Bühler. Structuring an Abstract Interpreter through Value and State Abstractions: EVA, an Evolved Value Analysis for Frama-C. Programming Languages [cs.PL]. Université de Rennes 1, 2017. English. <tel-01664726>

HAL Id: tel-01664726

<https://hal.archives-ouvertes.fr/tel-01664726>

Submitted on 15 Dec 2017

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 / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de
DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique
Ecole doctorale Matisse

présentée par

David Bühler

préparée à l'Unité Mixte de Recherche 6074 – IRISA
Institut de recherche en informatique et systèmes aléatoires
UFR Informatique Electronique (ISTIC)

**Structuring
an Abstract Interpreter
through Value and State
Abstractions :**

**EVA, an
Evolved Value Analysis
for Frama-C**

**Thèse soutenue à Rennes
le 15 mars 2017**

devant le jury composé de :

Antoine Miné

Professeur des universités – Université Pierre et Marie Curie / rapporteur

Mihaela Sighireanu

Maître de conférences – Université Paris Diderot / rapporteur

Thomas Jensen

Directeur de recherche – INRIA / examinateur

Yann Régis-Gianas

Maître de conférences – Université Paris Diderot / examinateur

Sandrine Blazy

Professeur des universités – Université de Rennes 1 / directrice de thèse

Boris Yakobowski

Ingénieur Chercheur – CEA LIST / co-directeur de thèse

*« Of all the communities available to us,
there is not one I would want to devote myself to,
except for the society of the true searchers,
which has very few living members at any time. »*

— Albert Einstein

*« We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty. »*

— Donald E. Knuth [Knu74]

This thesis is dedicated to the society of all those
who strive to be true searchers and artists.

*« How can one check a routine
in the sense of making sure that it is right?
[...]*

*In order to assist the checker,
the programmer should make assertions
about the various states that the machine can reach. »*

— Alan M. Turing, June 1949 [[Tur49](#)]

A respectful tribute to the memory of Alan Turing.

*[The C language has] the power of assembly language,
and the convenience of... assembly language.*

— Dennis Ritchie

Indeed.

— Anyone who ever tried to formally verify C programs.

RÉSUMÉ

La vérification formelle de programmes est devenue un enjeu majeur de l'informatique, à l'heure où des erreurs logicielles dans des systèmes critiques peuvent avoir des conséquences dramatiques. L'interprétation abstraite est une théorie générale d'approximation des sémantiques des langages de programmation, qui permet des analyses automatiques de programmes pour en détecter de façon certaine les comportements indésirables. Ces analyses reposent sur des abstractions d'une sémantique concrète, qui calculent une sur-approximation des comportements possibles d'un programme.

Cette thèse propose une nouvelle technique de composition modulaire entre les abstractions d'un interpréteur abstrait. L'idée principale en est l'organisation d'une sémantique abstraite suivant la distinction usuelle entre expressions et instructions. Les abstractions sont alors séparées entre abstractions de valeurs, en charge de la sémantique des expressions, et les abstractions d'états (ou domaines abstraits), en charge de la sémantique des instructions.

Cette adéquate hiérarchie guide les interactions entre abstractions durant l'analyse. Lors de l'interprétation d'une instruction, les états abstraits peuvent échanger des informations au moyen de valeurs abstraites, qui expriment des propriétés sur les expressions. Ces valeurs abstraites forment donc l'interface de communication entre les domaines, mais sont également des éléments canoniques de l'interprétation abstraite. Les outils standards de la théorie s'appliquent donc naturellement aux abstractions de valeurs. En particulier, elles peuvent elle-mêmes être composées par les techniques existantes, ouvrant la voie à plus d'interactions encore.

Cette thèse explore les possibilités offertes par cette nouvelle architecture des sémantiques abstraites. Elle décrit en particulier des stratégies efficaces pour le calcul de valeurs abstraites précises à partir des propriétés inférées par les domaines, et illustre les différents moyens d'interactions que ce système offre. Notre architecture comprend également une collaboration directe des différentes abstractions à l'émission des alarmes qui signalent les erreurs possibles d'un programme.

Ce système de composition des abstractions a été mis en œuvre dans [EVA](#), la nouvelle version de l'interpréteur abstrait de la plateforme Frama-C. [EVA](#) a été spécifiquement conçu pour faciliter l'introduction de nouvelles abstractions, et permettre des interactions riches entre ces abstractions. Grâce à son architecture modulaire et extensible, cinq nouveaux domaines abstraits ont pu être introduits dans l'analyseur en moins d'un an, améliorant ainsi tant ses capacités que sa précision.

ABSTRACT

The formal verification of programs is nowadays a crucial challenge for computer science, as software bugs in critical systems may lead to catastrophic outcomes. Abstract interpretation is a general theory of approximation of the semantics of programming languages, practically used to detect errors in programs. Automatic analyses can be derived by computing an over-approximation of the possible behaviors of a program, through abstractions of its concrete semantics.

This thesis proposes a new framework for the combination of multiple abstractions in the abstract interpretation theory. Its core concept is the structuring of the abstract semantics by following the usual distinction between expressions and statements. This can be achieved by a convenient architecture where abstractions are separated in two layers: value abstractions, in charge of the expression semantics, and state abstractions (or abstract domains), in charge of the statement semantics.

This design leads naturally to an elegant communication system where the abstract states, when interpreting a statement, interact and exchange information through abstract values, that express properties about expressions. While the values form the communication interface between states, they are also standard elements of the abstract interpretation framework. The communication system is thus embedded in the abstract semantics, and the usual tools of abstract interpretation apply naturally to value abstractions. For instance, different kinds of value abstractions can be composed through the existing methods of combination of abstractions, enabling even further interaction between the components of the abstract semantics.

This thesis explores the possibilities offered by this framework. We discuss efficient strategies to compute precise value abstractions from the properties inferred by abstract domains, and illustrate the means of communication between different state abstractions. Our architecture also features a direct collaboration for the emission of alarms that report the possible errors of a program.

The general system of abstractions combination has been implemented within [EVA](#), the new version of the abstract interpreter provided by the Frama-C platform. Thus, [EVA](#) enjoys a modular and extensible architecture designed to facilitate the introduction of new abstractions and to enable rich interactions between them. Thanks to this work, five new domains from the literature have been implemented in less than a year, enhancing both the scope and the precision of the analyzer.

REMERCIEMENTS

Ces travaux n'auraient jamais vu le jour si Boris Yakobowski, qui était alors mon directeur de stage, ne m'avait proposé de poursuivre une thèse, alors que, fidèle à moi-même, je n'y avais jamais réfléchi. Je veux aujourd'hui l'en remercier, car ces trois années (et demi) furent pour moi une aventure passionnante et fructueuse, et ce en grande partie grâce à lui. Je remercie tout aussi chaleureusement ma directrice, Sandrine Blazy, de m'avoir accompagné tout au long de cette thèse. Tous deux ont toujours été présents lorsque j'en ressentais le besoin, nonobstant leurs emplois du temps souvent (toujours ?) surchargés. Ils ont su me guider judicieusement et m'assister efficacement dans mes recherches, tout en me laissant une liberté d'action que j'ai beaucoup appréciée, et dont j'ai d'ailleurs largement profité.

Je remercie chaleureusement mes rapporteurs, Mihaela Sighireanu et Antoine Miné, d'avoir consacré de leur temps pour juger de mes travaux. L'intérêt qu'ils ont porté à ce manuscrit me fait grand honneur, et je leur suis reconnaissant des remarques pertinentes mais toujours bienveillantes qu'ils ont pu m'adresser pour m'aider à l'améliorer. Je remercie également Thomas Jensen et Yann Régis-Gianas d'avoir accepté de participer à mon jury. C'est un grand plaisir, en particulier, d'être amené à présenter mes recherches devant Yann, qui m'avait orienté vers le stage à l'origine de cette thèse et avait, déjà à l'époque, jugé de la qualité de mes travaux.

Je salue avec grande estime l'ensemble du Laboratoire de Sécurité du Logiciel, auquel je suis honoré et heureux d'appartenir. Je remercie chacun de ses membres avec lesquels j'ai travaillé ou partagé de bons moments. Pour leur aide précieuse, leur conseils avisés, leur soutien indéfectible, leur humour et leur bonne humeur, et pour toutes nos passionnantes conversations, qu'elles fussent ou non scientifiques, je rends grâce à François Bobot, André Maroneze, Virgile Prevosto, Florent Kirchner, Zakaria Chihani, Julien Signoles, Valentin Perelle, et j'en oublie encore... Je remercie en particulier Zakaria, pour sa relecture attentive de ce manuscrit. J'ai une pensée particulière pour Mounir Assaf, doctorant lorsque j'étais stagiaire, et qui, tant par ses qualités humaines que scientifiques, a été pour moi un exemple à suivre. Je remercie enfin les autres doctorants du laboratoire, pour toutes nos interactions, des plus dérisoires aux plus mémorables, durant ces trois années : Robin David, Allan Blanchard, Steven De Oliveira, Jean-Christophe Lechenet, Benjamin Farinier, Hugo Illous, Lionel Blatter et Vincent Botbol.

Je remercie les membres de l'équipe Celtique pour leur accueil chaleureux à chacune de mes incursions sur leurs terres. Au crépuscule

de cette thèse, je regrette n'avoir pas davantage profité de mon rattachement à l'université de Rennes pour les visiter plus souvent, et ainsi bénéficier de leur excellent niveau scientifique.

Je souhaite également remercier les assistantes de ces deux laboratoires, Frédérique Descreaux au LSL et Lydie Mabil à Celtique, qui m'ont à l'occasion sauvé de situations administratives inextricables. Je leur en suis très reconnaissant.

D'une façon générale, je remercie tous ceux avec qui mes rapports furent aussi divers qu'enrichissants.

Comme toute recherche scientifique, cette thèse n'est que la suite de travaux accomplis avant moi. Je ne saurais dresser une liste exhaustive des fondements, contributions et inspirations de mon travail personnel, mais je veux ici rendre hommage à tous ceux qui participent à faire avancer la science et propagent leurs savoirs, leurs idées et leurs expériences.

De plus, ce manuscrit ne couronne pas seulement trois ans de doctorat. Tout au long de mes études, j'ai eu la chance de rencontrer nombre de professeurs formidables, qui non seulement m'ont beaucoup appris, mais ont surtout su me faire partager leurs passions, dans des domaines très variés. Ma mémoire défaillante des noms et des visages m'empêchent de les citer précisément, mais je leur en suis particulièrement reconnaissant. J'espère qu'ils en ont (eu) conscience.

Enfin, mes derniers mots seront pour ma famille, que je remercie du fond du cœur : mon frère, que j'ai vu devenir au fil des années une personnalité remarquable, pour nos longues heures de jeux qui ont ponctué ma rédaction, et mes parents, *car c'étaient, au fond, d'excellents parents* [Aym39], pour avoir fait de moi ce que je suis. Il est impossible de quantifier ni de qualifier exactement ce que je vous dois, mais vous avez assurément su cultiver ma curiosité, ma sensibilité, mon amour des sciences et des arts. Vous m'avez toujours supporté, soutenu et encouragé dans chacun de mes choix. Je *suis* aujourd'hui grâce à vous. Merci pour tout.

CONTENTS

Résumé étendu en français	1
I CONTEXT	5
1 INTRODUCTION	7
2 ABSTRACT INTERPRETATION	17
2.1 Semantics of a Programming Language	17
2.1.1 Control-flow Graph and Denotational Semantics	17
2.1.2 A toy language	19
2.1.3 Syntax Simplifications	22
2.1.4 Collecting Semantics	23
2.2 Abstract Interpretation Principles	26
2.2.1 Main Concepts	26
2.2.2 Formalization	27
2.2.3 Lattices	28
2.2.4 Fixpoint Computation	30
2.2.5 Widening	32
2.2.6 Abstract Domains: Summary	33
2.3 Combination of Abstractions	34
2.3.1 Abstract Domains of the Literature	35
2.3.2 Direct Product	37
2.3.3 Reduced Product	38
2.3.4 Open Product	42
2.3.5 Communication through a Shared Language . .	42
2.3.6 Communication by Messages	43
2.3.7 Abstract Interpretation Based Analyzers	45
II EVA: A MODULAR ANALYZER FOR C	49
3 ARCHITECTURE OF THE ANALYZER	51
3.1 Overview of the EVA Structure	51
3.1.1 The Frama-C Platform	51
3.1.2 The Abstract Interpreter	53
3.1.3 Abstractions	53
3.2 A Modular Abstract Interpreter	55
3.2.1 Inner Workings of the Abstract Interpreter . . .	55
3.2.2 Combination of Abstractions	58
3.2.3 Instantiating the Abstractions	60
3.3 Structuring a Combination of Datatypes	60
3.3.1 Context and Motivation	60
3.3.2 Interface of a Combination	61
3.3.3 Polymorphic Keys	62
3.3.4 Naive Implementation	64
3.3.5 GADT Structure of a Datatype	64

3.3.6	Automatic Generation of the Accessors	67
3.4	Development and Contributions	69
3.4.1	Evolution of the Abstract Interpreter	69
3.4.2	Contributions	70
3.4.3	Development	71
4	SYNTAX AND SEMANTICS OF THE LANGUAGE	73
4.1	The C Language	73
4.1.1	The C Standard	74
4.1.2	C Intermediate Language	74
4.1.3	The C Spirit	75
4.2	A C-like Language	79
4.2.1	Syntax	80
4.2.2	Representation of Values in Memory	83
4.2.3	Validity of Pointers and Locations, Memories	86
4.2.4	Evaluation of Expressions in a Memory	87
4.3	A Concrete Semantics for Clite	89
4.3.1	Pointer Arithmetic and Memory Layout	89
4.3.2	Concrete States Independent of the Memory Lay- out	90
4.3.3	Concrete Semantics of Expressions	92
4.3.4	Concrete Semantics of Statements	96
III	ABSTRACT SEMANTICS OF EXPRESSIONS	99
5	VALUE ABSTRACTIONS	101
5.1	Alarms	101
5.1.1	Reporting Undesirable Behaviors	101
5.1.2	Alarms as ACSL Assertions for the End User	102
5.1.3	Set of Possible Alarms	103
5.1.4	Maps of Alarms	106
5.1.5	Propagating Alarms and Bottom Elements	110
5.2	Abstractions of Concrete Values	111
5.2.1	Concretization and Soundness of Value Abstrac- tions	112
5.2.2	Lattice Structure	114
5.2.3	Semantics of Values and Alarms	115
5.2.4	Abstraction of Constants	115
5.2.5	Abstraction of Operators	116
5.2.6	Abstractions of Memory Locations	118
5.3	The Cvalue Implementation	121
5.3.1	Basic Representation of Constant Values	121
5.3.2	Garbled Mix: a Representation of Not Constant Values	125
5.3.3	Forward Abstract Semantics of Cvalues	129
5.3.4	Meet of Garbled Mixes	132
5.3.5	Backward Propagators	137
6	EVALUATION OF EXPRESSIONS	141

6.1	Evaluation and Valuation	141
6.1.1	A Generic Functor	142
6.1.2	Abstract Domain Requirement	142
6.1.3	Valuations	144
6.1.4	Forward and Backward Evaluations	146
6.1.5	Atomic Updates of a Valuation	149
6.1.6	Complete Evaluations	153
6.1.7	Simplified Implementation	155
6.2	Forward and Backward Evaluation Strategies	157
6.2.1	Backward Propagation of Reductions	158
6.2.2	Forward Propagation of Reductions	163
6.2.3	Interweaving Forward and Backward Propagations	166
6.2.4	Evaluation Subdivision	173
IV	ABSTRACT SEMANTICS OF STATEMENTS	177
7	STATE ABSTRACTIONS	179
7.1	Collaboration for the Evaluation: Domain Queries . . .	180
7.1.1	Semantics of Dereference	180
7.1.2	Additional Query on any Expressions	184
7.1.3	Interaction through an Oracle	186
7.2	Backward Propagators	191
7.2.1	Backward Semantics of Dereference	191
7.2.2	Triggering New Reductions	194
7.3	Abstract Semantics of Statements	197
7.3.1	Abstract Semantics of Statements	198
7.3.2	Domain Product	201
7.3.3	Tracking Reductions	202
7.3.4	Related Works and Limitations	204
8	DOMAINS AND EXPERIMENTAL RESULTS IN EVA	209
8.1	The Cvalue Domain	209
8.1.1	Description	209
8.1.2	Integration in the EVA Framework	210
8.1.3	Performance Compared to VALUE	212
8.2	The Equality Domain	216
8.2.1	Dependences of an Expression	216
8.2.2	The Equality Abstract States and Queries	218
8.2.3	Interpretation of Assignments	221
8.2.4	Interpretation of Other Statements	227
8.2.5	Implementation	228
8.2.6	Experimental Results	229
8.3	Other New Domains in EVA	230
8.3.1	Binding to the APRON domains	230
8.3.2	The Symbolic Locations Domain	231
8.3.3	The Gauges Domain	231
8.3.4	Bitwise Abstractions	231

8.3.5	Experimental Results	231
8.3.6	Conclusion	232
V	ABSTRACT SEMANTICS OF TRACES	235
9	PREDICATED ANALYSES	237
9.1	Motivation	237
9.2	A Generic Abstract Interpretation Based Framework .	239
9.3	The Predicated Domain	241
9.3.1	Predicated Elements	241
9.3.2	Predicated Lattice	243
9.3.3	A Weaker Join	245
9.4	A Predicated Analysis	248
9.4.1	The Abstract Transfer Functions	248
9.4.2	Improving the Analysis: Avoiding Redundant Values	250
9.4.3	Propagating Unreachable States	253
9.4.4	Convergence of the Analysis	254
9.5	A Verified Soundness Proof	254
9.5.1	Prerequisites	254
9.5.2	Lattice Structure	255
9.5.3	Weak-Join	255
9.5.4	Analysis	256
9.6	Related Work	257
9.7	Experimental Results	260
9.7.1	Scope of the Current Implementation	260
9.7.2	Application on two Simple Domains	261
9.7.3	Results on Variables Initialization	262
9.7.4	Validation of the Optimizations	264
9.7.5	Experiments on Examples from the Literature .	266
9.8	Conclusion	267
VI	CONCLUSION	269
10	PERSPECTIVES	271
10.1	Summary	271
10.2	Future Works in EVA	272
10.3	Long-Term Perspectives	274
VII	APPENDIX	277
A	NOTATIONS SUMMARY	279
B	PROOFS	281
C	DEVELOPMENT FILES	285
	BIBLIOGRAPHY	290

RÉSUMÉ ÉTENDU EN FRANÇAIS

CONTEXTE

Il aura fallu moins de 80 années à l'informatique pour devenir un composant essentiel de nos sociétés. En 1936, Alan Turing pose les bases de l'informatique théorique ; dans les années qui suivent sont construites les premières machines précurseurs des ordinateurs modernes, désormais omniprésents dans nos vies quotidiennes. Nous nous sommes habitués à leurs avantages, et avons appris à gérer leurs occasionnels désagréments : les *bugs*, ou erreurs logicielles. Dans le même temps, les systèmes informatiques se sont également répandus dans l'industrie, et se retrouvent dans les appareils ménagers, les systèmes de transports, les équipements médicaux, les contrôleurs d'usine, les programmes spatiaux... Dans des systèmes critiques en particulier (tels que les robots médicaux, les voitures autonomes, l'aviation ou les centrales nucléaires), les conséquences d'une erreur logicielle peuvent se révéler dramatiques, en termes de vies humaines, d'impact environnemental ou de destructions matérielles. Cette observation met en évidence le besoin impérieux de méthodes fiables pour la détection des erreurs d'un programme, ou mieux encore, pour la preuve de leur absence.

Un moyen simple de découvrir les fautes d'un programme est de le tester, c'est-à-dire de l'exécuter en vérifiant si son comportement est bien conforme à ce qui en est attendu. Les tests sont couramment utilisés dans l'industrie, mais atteignent rapidement leur limites : les programmes informatiques dépendent généralement de leur contexte d'exécution et d'actions de l'utilisateur qui ne peuvent être exhaustivement essayés. Les méthodes de tests, aussi efficaces soient-elles pour détecter les erreurs, ne peuvent jamais en garantir l'absence.

C'est pour obtenir de telles garanties qu'ont été développées des méthodes formelles de raisonnement sur les programmes informatiques, fondées sur les mathématiques. Elles visent à établir une spécification logique des programmes, et à prouver que cette spécification est effectivement vérifiée par leurs implémentations. Elles reposent sur une sémantique formelle des langages de programmation, qui définit dans le monde mathématique la signification de chacun de leurs éléments syntaxiques. Un programme peut alors être vu et manipulé comme un objet mathématique sur lequel différentes propriétés peuvent être formulées et prouvées. Mais de nos jours, les programmes peuvent être composés de millions de lignes de code, et leur représentation mathématique est alors démesurément complexe... et ne peut être efficacement manipulée qu'au travers de l'informatique.

Cette approche connaît elle-même ses limitations : le théorème de Rice établit que toute propriété non triviale d'un langage de programmation est indécidable. Une conséquence directe de cet énoncé est l'impossibilité de développer un programme capable de déterminer automatiquement si un programme quelconque est erroné. Les outils de vérification formelle contournent cet obstacle en sacrifiant la complétude, se limitant à une certaine catégorie de programmes, ou requièrent une intervention humaine pour compléter leurs actions.

INTERPRÉTATION ABSTRAITE

Parmi les méthodes formelles, l'interprétation abstraite est une théorie générale d'approximation des sémantiques des langages de programmation. Elle découle de l'idée qu'il n'existe pas *une* sémantique universelle et idéale, mais de nombreuses façons de caractériser un langage. Puisqu'une sémantique concrète exacte se révèle généralement non calculable, l'interprétation abstraite propose de raisonner sur des sémantique *abstraites*, moins précises mais plus aisées à manipuler. De telles sémantiques permettent de calculer automatiquement une sur-approximation des comportements possibles d'un programme. Une propriété prouvée par la sémantique abstraite est alors vérifiée par toute exécution du programme. Les analyses fondées sur l'interprétation abstraite sont particulièrement efficaces pour démontrer l'absence d'opérations illégales menant à des échecs (divisions par zéro, accès invalides à la mémoire...). Néanmoins, les approximations opérées par une sémantique abstraite peuvent contrecarrer la vérification d'un programme.

La conception d'une sémantique abstraite —ou domaine abstrait— est délicate : celle-ci doit être suffisamment précise pour permettre la preuve des propriétés désirées, et suffisamment simple pour permettre l'analyse de larges programmes. Depuis l'introduction de l'interprétation abstraite par Patrick et Radhia Cousot à la fin des années 70, une large variété de domaines abstraits a été proposée dans la littérature. Chaque domaine possède ses avantages et ses inconvénients, offre un certain compromis entre précision et efficacité, et répond à différentes problématiques. L'une des forces de l'interprétation abstraite est la possibilité de composer plusieurs domaines abstraits en une seule analyse. En effet, la vérification d'un programme réel nécessite bien souvent la combinaison de différents domaines. De plus, les informations inférées par un domaine peuvent être utiles à un autre domaine dans son interprétation du programme. Pour atteindre une meilleure précision, un interpréteur abstrait doit donc mettre en œuvre une communication entre les différents domaines durant l'analyse d'un programme. Enfin, chaque domaine se montre plus ou moins efficace selon le programme considéré. Un analyseur

dispose d'un champ d'action d'autant plus large qu'il est modulaire, favorisant l'ajout, le retrait ou le remplacement de domaines abstraits.

CONTRIBUTIONS

Cette thèse propose un nouveau cadre pour la composition de domaines abstraits. L'idée principale en est l'organisation d'une sémantique abstraite suivant la distinction usuelle entre expressions et instructions, en cours dans la plupart des langages impératifs. Une expression exprime le calcul d'une valeur, alors qu'une instruction représente une action à exécuter. Un programme est alors une liste d'instructions à réaliser, définies au moyen d'expressions. La définition d'une sémantique abstraite peut se diviser entre abstractions de valeurs et abstractions d'états. Les abstractions de valeurs représentent les valeurs possibles d'une expression en un point donné, et assurent l'interprétation de la sémantique des expressions. Les abstractions d'états représentent les états machines qui peuvent se produire lors de l'exécution d'un programme, et permettent d'interpréter la sémantique des instructions.

De ce choix de conception découle naturellement un élégant système de communication entre abstractions. Lors de l'interprétation d'une instruction, les abstractions d'états peuvent échanger des informations au moyen d'abstractions de valeurs, qui expriment des propriétés à propos des expressions. Les valeurs forment donc une interface de communication entre états abstraits, mais sont également des éléments canoniques de l'interprétation abstraite. Ils peuvent donc eux-même être combinés par les moyens existants de composition d'abstractions, permettant encore davantage d'interactions entre les composants des sémantiques abstraites.

Cette thèse explore les possibilités offertes par cette nouvelle architecture des sémantiques abstraites. Nous décrivons en particulier des stratégies efficaces pour le calcul d'abstractions de valeurs précises à partir des propriétés inférées par les domaines, et nous illustrons les différentes possibilités d'interactions que ce système offre. L'architecture que nous proposons inclue également une collaboration directe des abstractions pour l'émission des alarmes qui signalent les erreurs possibles du programme analysé.

Nous proposons également un mécanisme permettant d'interagir avec les composants d'une combinaison générique de types OCaml. Nous utilisons des [GADT](#) pour encoder la structure interne d'une combinaison, et construisons automatiquement les fonctions d'injection et de projection entre le produit et ses composants. Cette fonctionnalité permet d'établir une communication directe entre les différentes abstractions d'un interpréteur abstrait.

Enfin, une dernière contribution de cette thèse est l'extension automatique de domaines abstraits à l'aide de prédicats logiques qui

évitent les pertes d’information aux points de jonction. De fait, lorsque plusieurs chemins d’exécution se rejoignent, un domaine abstrait doit représenter les comportements possibles de chacun des chemins, ce qui engendre souvent des pertes de précision. Pour remédier à cette limitation, nous proposons de propager un ensemble d’états abstraits, munis chacun d’un prédicat qui indique sous quelle condition l’état est valable. Contrairement à d’autres approches, notre analyse ne maintient pas une stricte partition des états abstraits, car les prédicats utilisés ne sont pas mutuellement exclusifs. Cette particularité rend possible des optimisations cruciales pour le passage à l’échelle de cette technique, confirmée par nos résultats expérimentaux sur un programme industriel généré.

MISE EN ŒUVRE AU SEIN DE FRAMA-C

FRAMA-C est une plateforme logicielle libre, extensible et collaborative, dédiée à l’analyse de programme C. Elle fournit un large éventail de fonctionnalités à travers plusieurs analyseurs qui exploitent différentes technologies pour vérifier des propriétés logiques sur des programmes. Ces propriétés peuvent être spécifiées par des annotations écrites dans un langage de spécification dédié. Depuis ses origines, FRAMA-C inclut un interpréteur abstrait nommé Value Analysis (ou simplement [VALUE](#)). Il calcule une sur-approximation des valeurs de chaque variable d’un programme, et émet une alarme en chaque point dont il échoue à prouver l’absence d’erreur à l’exécution. Cet analyseur est capable de traiter le sous-ensemble de C99 utilisé dans l’informatique embarquée, et a déjà été appliqué avec succès sur des codes industriels critiques. Néanmoins, [VALUE](#) ne bénéficie pas d’une architecture modulaire : il a été écrit autour de son domaine d’origine, et le fort couplage entre l’analyseur et ses abstractions rend difficile l’implémentation de nouveaux domaines abstraits.

L’ensemble du système de composition des abstractions proposé dans cette thèse a été mis en œuvre dans [EVA](#), la nouvelle version de l’interpréteur abstrait de FRAMA-C. [EVA](#) est une évolution majeure de [VALUE](#), et a été spécifiquement conçue pour faciliter l’introduction de nouvelles abstractions et permettre des interactions riches entre ces abstractions. Grâce à son architecture modulaire et extensible, cinq nouveaux domaines abstraits ont pu être introduit dans l’analyseur en moins d’un an, améliorant ainsi tant ses capacités que sa précision. Des efforts considérables ont également été consacrés à préserver les bonnes performances de l’analyseur. En particulier, le mécanisme de [GADT](#) décrit plus haut a été déployé pour maintenir certaines optimisations cruciales qui dépendent du domaine originel de [VALUE](#). Enfin, l’extension automatique de domaines abstraits à l’aide de prédicats disjonctifs a été implémentée en tant que plugin indépendant dans la plateforme FRAMA-C.

Part I

CONTEXT

INTRODUCTION

In the last decades, computer systems have become more and more pervasive in our everyday lives. We are now accustomed to use computers and smartphones everywhere, and also to face the occasional bugs in their operating systems, software components or online platforms. They may be a real annoyance for the users, but their resolution goes rarely beyond “rebooting the damn thing” in the worst case scenario. At the same time, but perhaps less visibly, computer systems also made inroads in the industry. “Embedded system” refers to a computer system integrated as part of a larger device, whose primary function is not computing. They are now widespread and essential in household appliance, transportation systems, medical equipments, factory controllers, space programs... This includes safety critical systems such as operating room machines, autonomous cars, flight-control systems in avionics and nuclear power plants managements. In such systems, a bug can lead to catastrophic outcomes in terms of human lives, environmental disasters or property damages. Most often, bugs cannot be easily circumvented in this context. This raises the need for efficient methodologies to detect bugs in computer programs beforehand, and even more importantly, to prove the absence of bugs in computer programs.

The execution of a program usually depends on some inputs that come from its context or from users actions. In order to hunt bugs, a program can be tested, by running it multiple times on various inputs, and by checking the behavior of each execution in compliance with some requirements. Testing methods are commonly used in industry, as they can be very effective for quickly discovering bugs in programs, especially at the earlier stages of their development. However, most programs accept an infinite set of possible inputs, and thus cannot be tested exhaustively in every possible configuration. Testing can then miss some bugs, and can never ensure the absence of bugs in a program.

To obtain stronger guarantees on program behaviors, we need to turn towards formal methods, that gather the techniques for reasoning about computer programs based on mathematical foundations. Formal methods aim at establishing a logical specification of programs, and at verifying that programs satisfy their specification. They rely on a formal semantics of the programming language used to write programs. The semantics gives a mathematical characterization of the meaning of each syntactic element of the language. Then, a program, which is a composition of these elements, can be seen as a

mathematical object on which properties can be formally stated and proved. The semantics is meant to describe precisely the possible behaviors of a program execution, according to its inputs. Various kinds of logical properties can be expressed about program behaviors. This thesis focuses on the safety property that the execution of a program can never cause a runtime error. A runtime error is a failure at the execution of an illegal operation (forbidden by the semantics of the programming language), such as a division by zero, a buffer overflow or an invalid memory access. The absence of runtime error does not ensure that the program behaves as expected by the programmer or by the user, but only that its execution does not crash. Other interesting properties include the termination of a program computation, or its functional correctness—that is proving that the output of the program meets some logical specification. However, establishing an exact specification of large and complex programs is often particularly challenging.

Even the mathematical representation of a program is generally huge and cumbersome: modern programs often consist of millions of lines of code, divided into various, nested and intricate components interacting with each other. This complexity prevents the proof of programs to be manageable manually. Instead, much efforts have been devoted to the mechanization of formal methods, and many tools have been developed to assist or even automatically achieve the verification of programs. However, this approach faces the barrier of the algorithmical undecidability of any non-trivial property about program semantics, stated by Rice’s theorem [Ric53]. A non-trivial property is neither true or false for every program. A semantic property is related to the formal semantics of the programming language (and not to its syntax). For such a non-trivial semantic property, there exists no algorithm (and thus no program) that decides for all program P whether P satisfies the property. A direct consequence of Rice’s theorem is the impossibility of a universal machine able to check in a finite time whether any program contains bugs. To circumvent this impossibility, formal verification tools sacrifice either completeness, by being limited to a specific class of programs, termination, which is another form of incompleteness, or complete mechanization, by eventually resorting to human interventions in order to overcome their inherent limitations.

The formal verification of programs has been an active research area from the early days of computer science, and various sets of techniques have sprung up since them. The current techniques include deductive methods, model checking and abstract interpretation, which is the subject of this thesis.

DEDUCTIVE VERIFICATION establishes the compliance of a program to its specifications as a collection of mathematical proof obligations, and discharges these obligations using **SMT** solvers (such

as Alt-Ergo, Z3 or CVC4) or interactive theorem provers (such as Coq or Isabelle). In this context, the specification of a program often consists of preconditions and postconditions for each of its functions. While deductive verification techniques relieve the user from the burden of most intermediate steps of a program proof, they cannot infer in general the inductive arguments required to handle loops. They thus rely on loop invariants that the user must provide.

MODEL CHECKING works on the model of a program, generally expressed as a Kripke structure or as a labelled transition system. The system is explored exhaustively to determine if all its possible sequences of states satisfy a given property. The property to be verified is generally expressed in temporal logic, and thus can be related to the execution traces of the program. Model checking has the advantage of being completely automatic, and can exhibit an erroneous execution trace when the program does not satisfy a property. However, designing a practical model of the program can be difficult, and the size of the model is critical: scalability issues prevent the use of model checking for large realistic programs.

ABSTRACT INTERPRETATION is a general theory for the analysis of computer programs by sound approximations of their semantics. A language semantics, defined as the most precise mathematical characterization of program behaviors, describes closely the execution of programs, but is generally not computable. The gist of abstract interpretation is to reason on relaxed *abstract* semantics, less precise but easier to handle. Analyses can be derived from a computable abstract semantics; they compute an over-approximation of the possible behaviors of a program, by interpreting it according to the given semantics.

Once given an abstract semantics, an abstract interpretation based analysis is completely automatic and can be applied to any program. However, the approximations made by the semantics may prevent the proof of the property to be verified. An analysis either proves the property despite its approximations, or does not state anything about it—due to overly wide approximations, or because the property is false. In other words, abstract interpretation may fail to prove the correctness of correct programs, but always detects incorrect programs.

The design of an abstract semantics—also called abstract domain—is always a delicate matter. Above all, an abstract semantics must be sound, by capturing all the possible behaviors of a program execution. This ensures the correctness of abstract interpretation based analyses: if a property can be proved within the abstract semantics, then the property is satisfied for every possible execution of the program. The soundness of an abstract semantics is usually guaranteed by relating

it to a more precise concrete semantics. Furthermore, an abstract semantics seeks to strike a balance between precision and efficiency, in order to enable the practical analysis of large and complex programs. It needs to be sufficiently subtle to prove the property to be verified, and tractable enough to scale on large codes. The abstract interpretation framework provides mathematical tools and methodologies to ensure the soundness of an abstract semantics, and the termination of the derived analyses.

The abstract interpretation has been introduced and developed by Patrick and Radhia Cousot in the late 1970s. Since then, much work has been conducted to design abstract domains suitable for the analysis of different classes of programs and for the proof of various families of properties. A wide variety of abstract domains have already been described in the literature; each one features different reasoning and approximations, and offers a different trade-off between accuracy and efficiency. A major asset of the abstract interpretation framework is the possibility to compose several abstract domains within a single analysis. Indeed, the verification of complex programs often requires joining the strengths of multiples abstract domains.

The principles of abstract interpretation have been applied to implement static analyzers that have already shown their industrial applicability to prove safety properties on critical codes. One of the most significant achievements of abstract interpretation remains the completely automatic proof of absence of runtime errors in the primary flight control software in the Airbus A340 and A380 airplanes by the Astrée analyzer [Ber+10]. Nevertheless, designing sound but precise abstract interpreters remains difficult. To enable accurate analyses on large classes of programs and properties, most analyzers implement a way of combining abstract domains where abstractions can be added, removed or replaced as needed. However, the combination of abstract domains is a challenge in itself. On the one hand, the domains must remain relatively independent: adding one domain should not require modifying the existing ones. On the other hand, they must also be able to cooperate by exchanging information, in order to achieve a better interpretation of the programs.

1.0.0.1 *The C Language*

In this thesis, we focus more specifically on the analysis of programs written in C. The C language, created by Thompson and Ritchie in the early 1970s as the development language of the Unix operating system, has become over the years one of the foremost programming languages in computer science. It remains nowadays among the most widely used programming language in the world, especially for embedded, safety-critical programs. *The C Programming Language* [KR78], written by Kernighan and Ritchie in 1978, was formerly regarded as the authoritative reference on C. Since 1989, the semantics of C is

officially specified by the successive versions of the C standard, published by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO); its current version is the C11 standard [C11]. However, these documents are all written in a natural language prose, without any mathematical formalization, and can only provide an informal description, with its inevitable share of ambiguities. This situation leads to some misunderstandings of the standard, and different visions (with subtle variations) of the C semantics coexist among programmers and compiler writers.

Moreover, the C language is oriented towards efficiency and portability. These features probably explain the success of the language, that offers strong performances on almost any existing hardware. They also make the language more bug-prone, by sacrificing the mathematical rigour and clarity needed to avoid errors. In particular, the language achieves its goals by underspecifications, and by exposing both low-level and high-level views of the memory.

UNDERSPECIFICATION The C standard contains hundreds of underspecifications, where the exact behavior of a specific construct is not precisely defined. In particular, the execution of any illegal operation (as a division by zero) may behave arbitrarily. This supports an efficient portability, as the compiler can choose the most practical way to handle these operations. This also makes bugs harder to detect and to understand, as their effects may vary and are generally unpredictable. This aspect of the C standard is often overlooked: some programmers rely on the common implementation of unspecified behaviors by current compilers, without any guarantee that their implementation choices will persist.

DUAL VIEWS OF THE MEMORY The C language features both low-level and high-level accesses to the memory (respectively via bit manipulations and typed expressions), and exposes the binary representation of high-level memory structures. Those dual views of the memory give more leeway to the programmers for implementing efficient programs, letting them choose the most convenient approach to address different algorithms. However, the interactions (and their restrictions) between the two models can be subtle and must be well understood. In particular, a commonly held view is that variable addresses and pointer values are simply integers, and can be handled accordingly. Even though the standard does not strictly legitimate this idea, a formal verification tool may choose to embrace it, in order to be able to verify the real-world programs that rely on this assumption.

While these features of the C language drive even more the need for the formal verification of C programs, they also are challenging to address precisely in sound and scalable analyzers.

1.0.0.2 *Frama-C*

The FRAMA-C platform is an extensible and collaborative framework dedicated to the analysis of C programs [Kir+15]. It provides a collection of interoperable analyzers, organized as plugins around a common kernel that centralizes information. Through its modular plugin architecture, FRAMA-C features a wide range of functionalities, and enables the user to exploit different techniques to prove properties on a program. The properties to be verified can be specified in the program as C annotation comments, written in a dedicated specification language. This formal language supports a wide variety of properties, and allows the user to write partial or complete specifications of functions.

FRAMA-C currently includes analyses based on abstract interpretation, deductive verification and dynamic checking. The work presented in this thesis is built upon the Value Analysis plugin (abbreviated as [VALUE](#)), that uses abstract interpretation techniques to over-approximate the values of the variables of a program. During its analysis, [VALUE](#) emits an *alarm* at each program point where it fails to prove the absence of an undefined behavior according to the C standard. It handles the subset of C99 commonly used in embedded softwares, and has already been successfully applied to verify safety-critical code. One of its key features is an intricate memory abstraction, able to represent efficiently and precisely both low-level and high-level concepts of the C memory model. However, the [VALUE](#) analyzer was written around a single abstract domain, resulting in a very tight coupling. So far, adding new abstract domains was not possible.

1.0.0.3 *Contributions*

This thesis presents the guiding principles, the design and the implementation of [EVA](#) [BBY17], the new abstract interpreter of the FRAMA-C platform. [EVA](#) stands for Evolved Value Analysis, and is a major evolution of the former [VALUE](#) analyzer. [EVA](#) overcomes the limitations of [VALUE](#) and features a modular and extensible architecture, designed to facilitate the implementation of new abstract semantics. Its main principle is to organize the internal abstractions by following the distinction between expressions and statements used in most imperative languages. An expression expresses the computation of a value (for instance, an integer) from a combination of constants, variables and operators. A statement represents an action to be carried out, such as the modification of a variable, or a jump in

another part of the code. Then, a program consists of a sequence of statements, which usually uses expressions to define their actions. The cornerstone of [EVA](#)'s architecture is the division of the abstract semantics between value and state abstractions. A value abstraction approximates the possible values of an expression, while a state abstraction represents the machine states that can occur at a program point during an execution. Value abstractions interpret the semantics of expressions, while state abstractions interpret the semantics of statements —whose actions are modeled on the abstract states.

This design of such an abstract interpreter leads quite naturally to a new communication system between abstractions. When interpreting a statement that contains some expressions, different state abstractions can exchange information through value abstractions of these expressions. This interaction system is elegant, as it is completely embedded in the abstract semantics of the analyzer: while the value abstractions act as a communication interface between state abstractions, they also are standard elements of the abstract interpretation framework. Thus, they can also be composed through the existing combination methods, enabling even more interactions between the components of the abstract semantics. In [EVA](#), both value and state abstractions are extensible.

The main contributions of this thesis are:

- a new framework for the combination and the interaction of multiple abstractions in the abstract interpretation theory. We define in this document the modular interfaces and the formal requirements that the abstractions must fulfill. We detail their means of communication, and prove that they do not contravene the soundness of the analysis. We also formalize a semantics and a cooperative emission mechanism for the alarms that report the possible bugs of a program.
- the implementation of this architecture within [EVA](#), the new open-source abstract interpreter of FRAMA-C. It has been used on various industrial case studies, and features a better precision and similar performances than the former abstract interpreter of FRAMA-C. Our design has also been validated by introducing different new abstractions in the analyzer.
- a mechanism to enable interacting with the components of a modular combination of OCaml types. We use [GADTs](#) [[JGo8](#)] to encode the inner shape of a combination, and automatically build injection and projection functions between a product of datatypes and its components. This mechanism has allowed us to maintain some crucial optimizations of the former [VALUE](#) analyzer that heavily rely on some specific abstraction within a modular combination.

- orthogonally, the automatic extension of abstract domains to track sets of disjunctive abstract states, each one being qualified with a predicate for which the state holds. This enhances the precision of an abstract semantics at join points, when several possible paths of a program execution meet. At these points, predicates preserve the information lost by the merge of abstract states. Unlike other approaches, the analysis does not maintain a strict partition of the abstract states, as the predicates we use are not mutually exclusive. This design enables some optimizations that are crucial for scalability, as confirmed by our experimental results on an industrial, generated Safety Critical Application Development Environment (SCADE) program. This mechanism has not been implemented within EVA, but as a new dedicated plugin of FRAMA-C that exploit the results of EVA. This work has been published in [BBY14] and [BBY16].

1.0.0.4 Overview of this Manuscript

This thesis is structured as follows.

Chapter 2 presents the mathematical foundations of our works: it formalizes the semantics of a programming language, and introduces the abstract interpretation framework. Special attention is devoted to the standard approaches for the combination of abstract semantics proposed in the literature.

Chapter 3 outlines the architecture and the core principles of EVA. It presents the hierarchy of abstractions, the inner working of the abstract interpreter, and the structuring of a modular product of OCaml types through GADT. These features do not depend on the analyzed language, and could be easily reused in another analyzer. This chapter also underlines the differences between VALUE and EVA.

Chapter 4 introduces the semantics of the C language, with a special focus on its underspecifications and its dual views of the memory. For the sake of simplicity, we formalize our works on a simplified language, smaller than C but with the same distinctive features. In particular, its pointer values are standard integers. This chapter defines this language and its semantics.

Chapter 5 is dedicated to the abstractions of expressions: the value abstractions, but also the alarms that report the possibly illegal operations on expressions. This chapter formalizes their interfaces and the soundness requirements of their semantics. It also presents the value abstractions currently available in EVA, focusing on their handling of pointer values as integers.

Chapter 6 shows how the abstract semantics of values and alarms enables the precise and complete evaluation of expressions, i.e. the computation of abstractions for an expression from abstractions of its subterms, and conversely. This chapter also presents the evaluation strategies implemented within EVA.

Chapter 7 is dedicated to the state abstractions —or abstract domains. It formalizes their interface and their formal specification, and proposes different examples to illustrate the opportunities for interactions between abstract domains using the evaluation of expressions into value abstractions. Finally, this chapter compares our communication system with the most relevant related works of the literature.

Chapter 8 describes the abstract domains that have been implemented within EVA so far, and presents the results of the experiments conducted to validate our works.

Chapter 9 presents the extension of an arbitrary abstract domain with conditional predicates to postpone the loss of information at join points. It formalizes the “predicated domains” and their semantics, and shows how to conduct efficient analyses over them. It is important to note that this last work has not been integrated into EVA, but as a new plugin of FRAMA-C.

Chapter 10 concludes this thesis. It summarizes the results achieved in this thesis, and proposes some ideas of future works, to further improve the EVA analyzer.

ABSTRACT INTERPRETATION

This thesis is based on abstract interpretation, a general theory for the sound approximation of the semantics of computer programs. Its core concept is that there is no innate, universal or ideal semantics to characterize program behaviors. Instead, a wide variety of formal semantics can be designed to this end, and one should choose carefully the most appropriate one to prove a specific property on a given program. Abstract interpretation links a very precise, but generally non-computable, concrete semantics to abstract ones – the abstract semantics being sound approximations of the concrete one. The abstract semantics are thus conservative: the absence of errors in an abstract semantics ensures the absence of errors in the concrete semantics.

This chapter describes our mathematical representation of computer programs, and formalizes more specifically the collecting semantics of a simplistic programming language. Then, it introduces the mathematical foundations of abstract interpretation, and defines the concepts and the notations used in this thesis. It finally presents some well-known abstract semantics of the literature, and the different ways to compose them in the abstract interpretation framework.

2.1 SEMANTICS OF A PROGRAMMING LANGUAGE

In computer science, the formal methods gather the techniques based on logic and mathematics for the specification, the development and the verification of algorithms and computer programs. In this context, a formal verification is a mathematical proof of the correspondence between a program and a logical specification. The first step towards the formal verification of programs is the mathematical characterization of the programming language used to write them. A programming language is a formal notation designed to express algorithms by encoding sequences of instructions that a machine (a computer) can execute. It is described by its syntax —the definition of the valid sequences of characters of the language— and by its semantics —the meaning of its syntactic entities, i.e. their effect when they are executed by the machine.

2.1.1 *Control-flow Graph and Denotational Semantics*

The behavior of a program (of its execution by a machine) can be formally described as a sequence of machine states. We call *statements* the smallest syntactic elements of the language that express an action

to be carried out. The execution of a statement can alter the current state: for instance, assignments modify the value of a variable. We call control structures the syntactic parts of the language that specify the order of statements execution. In this thesis, programs are represented as *control-flow graphs* that explicitly encode the control structure of a program; the edges are labeled by statements.

The statement semantics of a language can be defined using three main kinds of formal semantics. An *operational* semantics closely describes the behavior of a construct execution through a transition system between the program states. Transitions can be defined as atomic execution steps (in small-step semantics), or inductively as sequences of computational steps (in big-step semantics). An *axiomatic* semantics establishes logical implications between assertions valid before a statement and assertions valid after it. The assertions are logical predicates describing the program states, closely related to Hoare logic. A program and the property to be verified can be translated into a logic formula, and proving that the program satisfies the property is reduced to proving the validity of the formula. Finally, a *denotational* semantics formalizes the meaning of the language syntax through mathematical objects (called denotations). This is the most abstract definition of a language semantics, independent of its concrete implementation. This thesis uses denotational semantics as they are easier to manipulate in mathematical proofs. For instance, we define the semantics of statements as functions on the program states.

Definition 1 (Control-flow graph). A control-flow graph is a directed labeled graph $G = (N, init, final, \Sigma, I, \Omega, T)$ where:

- the finite set of nodes N is the set of program points. Among them, $init \in N$ and $final \in N$ are respectively the initial and the ending node of the program.
- Σ is a set of states. Among them, $I \subseteq \Sigma$ is the set of possible initial states and $\Omega \subseteq \Sigma$ is a set of erroneous states.
- T is a set of labeled edges, namely transitions (n, f, m) between two nodes $n \in N$ and $m \in N$ with a function f such that:

$$f : \Sigma \setminus \Omega \rightarrow \Sigma \uplus \emptyset$$

f is the denotation of a language statement, modeling its effect on a non-erroneous program state. It computes a new (possibly erroneous) state as the result of the execution of the statement, or the special value \emptyset if the statement cannot be executed from the argument state.

By convention, program points (nodes) are represented by natural integers $N \subseteq \mathbb{N}$ and the initial node is always 0.

The erroneous states are states that a correct program should never

reach. They can correspond to bugs or crashes of a program execution.

Henceforth, we identify programs and control-flow graphs. A control-flow graph describes the possible executions of a program as sequences of pairs of a state in Σ and a node in N . An execution starts with an initial state in I at the node *init*, and then follows the transitions given by the edges of the graph. It is worth noting that the graph structure can encode non-determinism, if multiple transitions can be chosen for a state at a point. An execution stops with success when reaching the final node *final*, and stops by failure if reaching an erroneous state. An execution may also never end, either through an infinite sequence of states, or by reaching a node without any transition to apply.

The representation of a program execution as a sequence of states and nodes is called a *trace*.

Definition 2 (Traces). Let $G = (N, \text{init}, \text{final}, \Sigma, I, \Omega, T)$ be a control-flow graph. For $K = \mathbb{N}$ or $K = \{0, \dots, \ell\}$, the sequence $(S_k, n_k)_{k \in K} \in (\Sigma \times N)^K$ is a trace of G if:

$$\begin{cases} S_0 \in I \quad \wedge \quad n_0 = \text{init} \\ \forall k \in K \setminus \{0\}, \exists (n_{k-1}, f, n_k) \in T, S_k = f(S_{k-1}) \neq \emptyset \end{cases}$$

The finite sequence $(S_k, n_k)_{k \in \{0, \dots, \ell\}}$ is a complete execution of G if, on top of that, $n_\ell = \text{final}$. Infinite sequences are also complete executions of G .

The sequence $(S_k, n_k)_{k \in \{0, \dots, \ell\}}$ is an erroneous execution of G if $S_\ell \in \Omega$.

The set of the possible behaviors of a program execution is then described by all its traces. This is the *trace semantics* of a program.

Definition 3 (Trace semantics). The trace semantics of a program G is the set of its traces, denoted \mathcal{T}_G .

2.1.2 A toy language

This chapter is based on a simplified language that we call Toy. A Toy program operates on a finite set \mathcal{X} of integer variables, whose values can change during the execution. It is thus natural to define the states of a program as the set of environments $\rho : \mathcal{X} \rightarrow \mathbb{N}$ that link each program variable to an integer, augmented with a single erroneous state Ω —as there is only one erroneous state, we identify the set of erroneous states with this state.

$$\Sigma = (\mathcal{X} \rightarrow \mathbb{N}) \uplus \Omega$$

Figure 2.1 presents the syntax of the Toy language, divided into statements and expressions. Expressions are either integer constants,

Variables:	$x, y, z \in \mathcal{X}$	
Expressions:	$e ::= n$	$n \in \mathbb{Z}$ <i>integers</i>
	$ x$	$x \in \mathcal{X}$ <i>variables</i>
	$ e + e \mid e - e \mid e \times e \mid e \div e$	<i>arithmetic</i>
	$ e = e \mid e \neq e \mid e < e \mid e \leq e$	<i>comparisons</i>
Statements:	$stmt ::= x := e$	<i>assignment</i>
	$ e == 0?$	<i>test</i>
	$ skip$	<i>identity</i>

Figure 2.1: Syntax of the Toy language

variables or arithmetic operations and comparisons between subexpressions. Their mathematical meaning should be clear. Statements are:

- assignments $x := e$, whose effect is to change the value of the given variable x into the value of the expression e in the current environment.
- test filters $e == 0?$ that enable the transition only if the expression e has the value 0. In other words, it blocks the execution for the environments in which e is not zero.
- skip statements, which have no effect on the program states.

Again, their semantics is really standard. Figure 2.2 formally defines the semantics of statements and expressions as mathematical functions.

The evaluation of an expression e is a function $\llbracket e \rrbracket$ from environments to integers or to the erroneous state.

$$\llbracket e \rrbracket : (\mathcal{X} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \uplus \Omega)$$

The evaluation of a variable is its value in the environment; the evaluation of arithmetic operations follows the integer arithmetics; the evaluation of a comparison is 1 if the comparison holds, and 0 otherwise. The evaluation of a division fails if the evaluation of the divisor is 0.

According to the definition 1 of control-flow graph, the semantics of a statement s is a function:

$$\llbracket s \rrbracket : (\mathcal{X} \rightarrow \mathbb{N}) \rightarrow \Sigma \uplus \emptyset$$

Given an environment ρ , we denote by $\rho[x \leftarrow v]$ the environment ρ in which x has the image v .

$$\rho[x \leftarrow v](y) = \begin{cases} v & \text{if } y = x \\ \rho(y) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\llbracket n \rrbracket(\rho) &\triangleq n \\
\llbracket x \rrbracket(\rho) &\triangleq \rho(x) \\
\llbracket e_1 \diamond e_2 \rrbracket(\rho) &\triangleq \begin{cases} \Omega & \text{if } \llbracket e_i \rrbracket(\rho) = \Omega \\ \llbracket e_1 \rrbracket(\rho) \diamond \llbracket e_2 \rrbracket(\rho) & \text{otherwise} \end{cases} \quad \forall \diamond \in \{+, -, \times\} \\
\llbracket e_1 \diamond e_2 \rrbracket(\rho) &\triangleq \begin{cases} \Omega & \text{if } \llbracket e_i \rrbracket(\rho) = \Omega \\ 1 & \text{if } \llbracket e_1 \rrbracket(\rho) \diamond \llbracket e_2 \rrbracket(\rho) \\ 0 & \text{otherwise} \end{cases} \quad \forall \diamond \in \{=, \neq, <, \leq\} \\
\llbracket e_1 \div e_2 \rrbracket(\rho) &\triangleq \begin{cases} \Omega & \text{if } \llbracket e_i \rrbracket(\rho) = \Omega \vee \llbracket e_2 \rrbracket(\rho) = 0 \\ \llbracket e_1 \rrbracket(\rho) / \llbracket e_2 \rrbracket(\rho) & \text{otherwise} \end{cases} \\
\llbracket x := e \rrbracket(\rho) &\triangleq \begin{cases} \Omega & \text{if } \llbracket e \rrbracket(\rho) = \Omega \\ \rho[x \leftarrow \llbracket e \rrbracket(\rho)] & \text{otherwise} \end{cases} \\
\llbracket e == 0? \rrbracket(\rho) &\triangleq \begin{cases} \Omega & \text{if } \llbracket e \rrbracket(\rho) = \Omega \\ \rho & \text{if } \llbracket e \rrbracket(\rho) = 0 \\ \emptyset & \text{otherwise} \end{cases} \\
\llbracket skip \rrbracket(\rho) &\triangleq \rho
\end{aligned}$$

Figure 2.2: Denotational semantics of Toy

The semantics of an assignment $x := e$ changes an environment ρ into the environment $\rho[x \leftarrow v]$, where v is the result of the evaluation of e in ρ . The semantics of a test $e == 0?$ in an environment ρ is the identity if e evaluates to 0, and blocks otherwise. Both semantics lead to the error state if the evaluation of e is the error state. Finally, the semantics of the skip statement is the identity function.

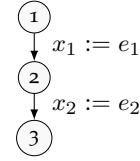
We henceforth use the statements to denote their semantics in program graphs: a transition is (n, stmt, m) where n and m are two program points, and stmt a language statement.

2.1.3 Syntax Simplifications

Encoding the control structures by control-flow graphs allows us to focus only on the statement semantics, and greatly simplifies the formalization of the language. However, graphs are not a very convenient way for humans to write algorithms or programs. We use instead the standard control structures of imperative programming languages—sequences of statements, conditional branches through *if* instructions and loops—as syntactic sugar for the control-flow graphs that they represent. The connection between those syntactic structures and control-flow graphs is given below. In this thesis, most code examples are written using this C-like syntax.

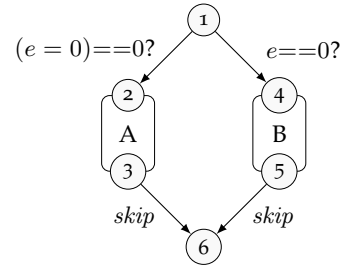
SEQUENCES:

```
1:   $x_1 := e_1$ ;
2:   $x_2 := e_2$ ;
3:  ...
```



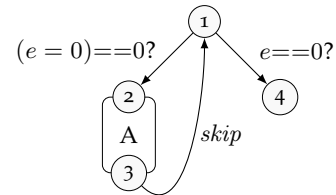
CONDITIONAL BRANCHES: the condition $e \neq 0$ induced by an *if*(e) statement is translated into the test filter $(e = 0) == 0?$, which is equivalent.

```
1:  if ( $e$ ) {
2:    [A]
3:  } else {
4:    [B]
5:  }
6:  ...
```



LOOP:

```
1:  while ( $e$ ) {
2:    [A]
3:  }
4:  ...
```



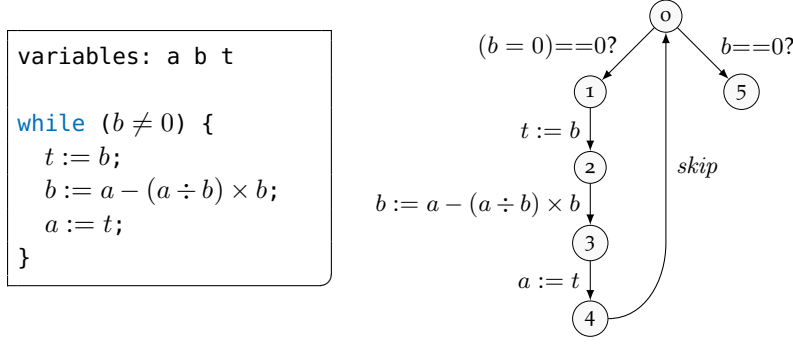


Figure 2.3: Euclidean algorithm

Example 1. Figure 2.3 presents an optimized implementation of the oldest known non-trivial algorithm: the Euclidean algorithm. It operates on three variables a , b and t ; the initial node is 0 and the final node is 5. As the modulo operation $a \bmod b$ does not exist in the language, it is translated into $a - (a \div b) \times b$. When an execution reaches this final node, the variable a has the value of the greatest common divisor (gcd) of the initial values of a and b .

2.1.4 Collecting Semantics

The definition 1 of a program includes a set of erroneous states. They describe undesirable behaviors that should not happen during a program execution and that we aim to prevent. In the Toy language for instance, a program reaches the erroneous state when a division by zero occurs, which is mathematically undefined. In this thesis, we are interested in formally proving the property that a given program is free of undesirable behaviors. This means that an execution of the program cannot reach an erroneous state. This property can be seen as a global invariant of all possible program executions. In the trace semantics, a program satisfies this invariant if none of its traces reaches an erroneous state. We then say that the program is correct.

$$\nexists (S_k, n_k)_{k \in \{0, \dots, n\}} \in \mathcal{T}_G, S_n \in \Omega$$

However, we do not need the full expressivity of the trace semantics to formalize this property. We are only interested in the states that a program execution can reach, but not in the connection between the successive states. We thus can overlook the exact traces of a program and reason only on the set of reachable states. This is our choice of the *collecting semantics* of a program, that connects each program point to the set of possible states at this point.

Definition 4 (Collecting semantics). The collecting semantics of a program $G = (N, init, final, \Sigma, I, \Omega, T)$, denoted \mathcal{C}_G , is the function from nodes N to states Σ defined as:

$$\begin{aligned} \mathcal{C}_G : N &\rightarrow \Sigma \\ \mathcal{C}_G(n) &\triangleq \{S \in \Sigma \mid \exists t \in \mathcal{T}_G, (S, n) \in t\} \end{aligned}$$

The collecting semantics is strictly less expressive than the trace semantics. The trace semantics can be used to ensure the termination of a program G (if all traces of \mathcal{T}_G are finite), or to express properties that relate the final state of an execution with its initial state. In the program G of example 1, the correction of the Euclidian algorithm can be stated by the following property:

$$\forall (\rho_k, pc_k)_{k \in \{0, \dots, n\}} \in \mathcal{T}_G, pc_n = final \Rightarrow \rho_n(a) = \gcd(\rho_0(a), \rho_0(b))$$

The simpler collecting semantics, although unable to express such properties, allows however a straightforward definition of the correctness of programs.

Definition 5. A program G is correct if $\forall n \in N, \mathcal{C}_G(n) \cap \Omega = \emptyset$.

While the collecting semantics is really convenient to our needs, it is built upon the trace semantics. Let us rather define the collecting semantics as the solution of equations between the sets of reachable states at each node of a program, using the transition system of the graph. To simplify the equations, we assume that no transition ends at the initial node. This property ensures that $\mathcal{C}_G(init)$ is exactly the set of initial states of the program G . Any program can be turned into a graph that satisfies this property by adding a new initial node and a skip statement from it to the previous initial node.

Theorem 1. Let $G = (N, init, final, \Sigma, I, \Omega, T)$ be a program such that $\nexists (n, f, init) \in T$. The collecting semantics \mathcal{C}_G is the smallest solution of the following system of equations:

$$\begin{aligned} C(init) &= I \\ \forall n \in N \setminus \{init\}, \quad C(n) &= \bigcup_{(m, stmt, n) \in T} \{ \llbracket stmt \rrbracket(S) \mid S \in C(m) \} \end{aligned} \quad (2.1)$$

Proof. Let us prove first that \mathcal{C}_G is a solution of this equation system. By definition 4 of the collecting semantics:

$$S \in \mathcal{C}_G(n) \Leftrightarrow \exists (t_i)_{i \in K} \in \mathcal{T}_G \wedge \exists k \in K, t_k = (S, n)$$

By definition 2 of traces:

$$\begin{aligned} k = 0 &\Rightarrow (S, n) = t_0 \Rightarrow S \in I \wedge n = init \\ k > 0 &\Rightarrow \begin{cases} S' \in \mathcal{C}_G(n') \\ \exists (n', stmt, n) \in T, S = \llbracket stmt \rrbracket(S') \end{cases} \quad \text{with } (S', n') = t_{k-1} \end{aligned}$$

As $\nexists(n, f, \text{init}) \in T$, if $k > 0$ then $n \neq \text{init}$. Thus, $\mathcal{C}_G(\text{init}) \subseteq I$ and as I is defined as the initial states at the program point init , we naturally obtain $\mathcal{C}_G(\text{init}) = I$.

Conversely to the $k > 0$ case, we assume a state $S' \in \mathcal{C}_G(n')$ and a transition $(n', \text{stmt}, n) \in T$ such that $S = \llbracket \text{stmt} \rrbracket(S')$. Then:

$$\exists(t_i)_{i \in K} \in \mathcal{T}_G \wedge \exists k \in K, t_k = (S', n')$$

We define a new sequence (t'_i) as:

$$\begin{aligned} \forall i \in \{0, \dots, k\} \quad t'_i &= t_i \\ t_{k+1} &= (S, n) \end{aligned}$$

This sequence is a trace of G , as (t_i) is a trace of G and by hypothesis $S = \llbracket \text{stmt} \rrbracket(S')$ for the transition (n', stmt, n) . Thus, $S \in \mathcal{C}_G(n)$.

$$S \in \mathcal{C}_G(n) \Leftrightarrow \exists S' \in \mathcal{C}_G(n') \wedge \exists(n', \text{stmt}, n) \in T, S = \llbracket \text{stmt} \rrbracket(S')$$

This ensures finally:

$$\forall n \in N \setminus \{\text{init}\}, \quad \mathcal{C}_G(n) = \bigcup_{(m, \text{stmt}, n) \in T} \{ \llbracket \text{stmt} \rrbracket(S) \mid S \in \mathcal{C}_G(m) \}$$

Let us now prove that \mathcal{C}_G is the *smallest* solution of the equation system via the following stronger lemma. \square

Lemma 1. Let $G = (N, \text{init}, \text{final}, \Sigma, I, \Omega, T)$ be a program such that $\nexists(n, f, \text{init}) \in T$. The collecting semantics \mathcal{C}_G is the smallest solution of the following system of equations:

$$\begin{aligned} C(\text{init}) &\supseteq I \\ \forall n \in N \setminus \{\text{init}\}, \quad C(n) &\supseteq \bigcup_{(m, \text{stmt}, n) \in T} \{ \llbracket \text{stmt} \rrbracket(S) \mid S \in C(m) \} \end{aligned} \quad (2.2)$$

In other words, if $X : N \rightarrow \mathcal{P}(\Sigma)$ is a solution of this system of equations, then $\forall n \in N, \mathcal{C}_G(n) \subseteq X(n)$.

Proof. Let X be a solution of 2.2. Then $X(\text{init}) \supseteq I$. Let $n \in N$ different from init . We need to prove that $\mathcal{C}_G(n) \subseteq X(n)$. Let S be a state of $\mathcal{C}_G(n)$.

$$\exists(S_i, n_i)_{i \in K} \in \mathcal{T}_G \wedge \exists k \in K, S = S_k \wedge n = n_k$$

By definition 2 of traces:

$$\begin{cases} S_0 \in I \quad \wedge \quad n_0 = \text{init} \\ \forall k \in K \setminus \{0\}, \exists(n_{k-1}, \text{stmt}, n_k) \in T, S_k = \llbracket \text{stmt} \rrbracket(S_{k-1}) \end{cases}$$

We can easily prove by induction that $\forall k \in K, S_k \in X(n_k)$. The base case is immediate: $S_0 \in I \subseteq X(n_0)$. We assume $S_{k-1} \in X(n_{k-1})$. There exists a transition $(n_{k-1}, \text{stmt}, n_k) \in T$ such that $n_k \neq \text{init}$ and $S_k = \llbracket \text{stmt} \rrbracket(S_{k-1})$. Equation 2.1 ensures that $S_k \in X(n_k)$.

In particular, $(S, n) \in (S_i, n_i)_{i \in K}$ and thus $S \in X(n)$.

We have thus proved $\mathcal{C}_G(n) \subseteq X(n)$. \square

Although suitable for expressing the property we aim at proving, the collecting semantics is usually not computable. The semantics of the Toy language manipulates infinite sets of states, and even the semantics of the Euclidian algorithm given in Figure 2.3 would not be easy to formalize. On a real machine, the number of possible states is finite but remains far too oversized to be used directly as means of proof. We now need to work on more practical *approximations* of this precise semantics.

2.2 ABSTRACT INTERPRETATION PRINCIPLES

The abstract interpretation [CC77a; CC79b; CC92a; Cou81; Cou78] is a fundamental theory and a practical framework for the realistic approximation of the semantics of programs. This section presents and formalizes its principles. It explains how to ensure that an approximated semantics is correct according to the collecting semantics, and how to guarantee the termination of an abstract interpretation based analysis.

2.2.1 Main Concepts

A language semantics, defined as a precise mathematical characterization of program executions, is generally not computable. Such semantics—the collecting semantics, for instance—are called *concrete* semantics. The gist of abstract interpretation is to reason on a relaxed *abstract* semantics, designed to be easier to handle. An abstract semantics is an approximated characterization of programs executions.

Collecting semantics link each program point to a set of reachable states. The states used to define a concrete semantics are called concrete states. On the other hand, an abstract semantics operates on abstract states. An abstract state represents a set of concrete states, and an abstract collecting semantics links each program point to one abstract state. The abstract interpretation theory provides a methodology to ensure that an abstract semantics is computable and *sound*. An abstract semantics is sound if it captures all the possible behaviors (all the executions) of a program. This means that the abstract state at a program point must represent at least all the reachable states of the collecting semantics. In a sound abstract semantics, an abstract state expresses a property (an invariant) that holds in all reachable states. In particular, if the abstract states of all program points exclude the erroneous states, the soundness of the abstract semantics guarantees that no erroneous state is a reachable state: the program is then proved correct.

However, the abstract semantics is an over-approximation of the concrete semantics. If the representation of an abstract state includes an erroneous state, we cannot conclude that this is actually a reach-

able state. Designing an abstract semantics always involves a continuing trade-off between accuracy and efficiency: it must be precise enough to exclude the erroneous states on correct programs, but its computation must be tractable and scale on large programs.

2.2.2 Formalization

The collection of abstract states over which operates an abstract semantics is called an *abstract domain*. The function that links each abstract state to the set of concrete states it represents is called the *concretization* of the domain.

Definition 6 (Abstract domain). An abstract domain is a set \mathbb{D} of abstractions and a concretization function γ from elements of the domain to sets of concrete states.

$$\gamma : \mathbb{D} \rightarrow \mathcal{P}(\Sigma)$$

We said that an abstract state $d \in \mathbb{D}$ abstracts or represents the concrete states of $\gamma(d)$.

Traditionnally, the elements e of an abstract semantics are denoted with a sharp note e^\sharp . In this thesis, we prefer the octothorpe $e^\#$.

Definition 7 (Abstract semantics). Let $G = (N, init, final, \Sigma, I, \Omega, T)$ be a program, and \mathbb{D} be an abstract domain. A sound abstract semantics of G is a function $\mathcal{C}_G^\# : N \rightarrow \mathbb{D}$ such that:

$$\forall n \in N, \mathcal{C}_G(n) \subseteq \gamma(\mathcal{C}_G^\#(n))$$

An abstract semantics on an abstract domain can be defined through functions that over-approximate the concrete semantics of the language statements. Such functions—one for each statement kind—are called *transfer functions*. An abstract domain must also be equipped with an over-approximation of the union of concrete states.

Definition 8 (Transfer function). A sound transfer function for a statement $stmt$ is a function $\llbracket stmt \rrbracket^\#$:

$$\begin{aligned} \llbracket stmt \rrbracket^\# : \mathbb{D} &\rightarrow \mathbb{D} \\ \forall S^\# \in \mathbb{D}, \llbracket stmt \rrbracket(\gamma(S^\#)) &\subseteq \gamma(\llbracket stmt \rrbracket^\#(S^\#)) \end{aligned}$$

Definition 9 (Inclusion and join). A sound approximation of the inclusion of concrete states is a relation \sqsubseteq between abstract states such that:

$$\forall (S_1^\#, S_2^\#) \in \mathbb{D}, S_1^\# \sqsubseteq S_2^\# \Rightarrow \gamma(S_1^\#) \subseteq \gamma(S_2^\#)$$

A sound approximation of the union of concrete states is a join operation \sqcup between abstract states such that:

$$\forall (S_1^\#, S_2^\#) \in \mathbb{D}, \gamma(S_1^\#) \cup \gamma(S_2^\#) \subseteq \gamma(S_1^\# \sqcup S_2^\#)$$

Theorem 2. Let $G = (N, init, final, \Sigma, I, \Omega, T)$ be a program such that $\#(n, f, init) \in T$, and D be an abstract domain. Any solution of the following system of equations is a sound abstract semantics of G :

$$\begin{aligned} \gamma(X(init)) &\supseteq I \\ \forall n \in N \setminus \{init\}, \quad X(n) &\sqsubseteq \bigsqcup_{(m, \text{stmt}, n) \in T} (\{\text{stmt}\}^\#(X(m))) \end{aligned} \quad (2.3)$$

Proof. Let $X : N \rightarrow \mathbb{D}$ be a solution of 2.3. Let $C : X \rightarrow \mathcal{P}(\Sigma)$ defined as $\forall n \in N, C(n) = \gamma(X(n))$. Then we have:

$$\begin{aligned} C(init) &\supseteq I \\ \forall n \in N \setminus \{init\}, \quad X(n) &\sqsubseteq \bigsqcup_{(m, \text{stmt}, n) \in T} (\{\text{stmt}\}^\#(X(m))) \\ \Rightarrow C(n) &\supseteq \gamma(\bigsqcup_{(m, \text{stmt}, n) \in T} (\{\text{stmt}\}^\#(X(m)))) \\ \Rightarrow C(n) &\supseteq \bigcup_{(m, \text{stmt}, n) \in T} (\gamma(\{\text{stmt}\}^\#(X(m)))) \\ \Rightarrow C(n) &\supseteq \bigcup_{(m, \text{stmt}, n) \in T} (\{\text{stmt}\}(\gamma(X(m)))) \\ \Rightarrow C(n) &\supseteq \bigcup_{(m, \text{stmt}, n) \in T} (\{\text{stmt}\}(C(m))) \end{aligned}$$

C is thus a solution of the system of equations 2.2, and by lemma 1:

$$\forall n \in N, \mathcal{C}_G(n) \subseteq C(n) = \gamma(X(n))$$

Any solution of 2.3 is a sound abstract semantics of G , according to definition 7. \square

We will focus now on the conditions ensuring that the system of equations has a solution, and how to efficiently compute it.

2.2.3 Lattices

The abstractions used in abstract interpretation have generally a lattice structure. We introduce here the standard notions of lattices in order theory and as an algebraic structure.

Definition 10 (Partially ordered set). A partial order over a set E is a binary relation \sqsubseteq that is reflexive, antisymmetrical and transitive:

- $\forall x \in E, x \sqsubseteq x$ (reflexivity)
- $\forall (x, y) \in E^2, (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$ (antisymmetry)
- $\forall (x, y) \in E^2, (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$ (transitivity)

A set with a partial order (E, \sqsubseteq) is called a *partially ordered set* (or *poset*).

Definition 11 (Lower and upper bounds). Let (E, \sqsubseteq) be a partially ordered set, and $S \subseteq E$ a subset of it. An *upper bound* of S is an element $u \in E$ such that $\forall x \in S, x \sqsubseteq u$. An upper bound b of S is its *least upper bound* if $b \sqsubseteq b'$ for each upper bound b' of S . A *lower bound* of S is an element $l \in E$ such that $\forall x \in S, l \sqsubseteq x$. A lower bound l of S is its *greatest lower bound* if $l' \sqsubseteq l$ for each lower bound l' of S .

Definition 12 (Lattice). A *lattice* $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$ is a partially ordered set $(\mathcal{L}, \sqsubseteq)$ where each pair $(x, y) \in \mathcal{L}^2$ has a least upper bound, denoted by $x \sqcup y$, and a greatest lower bound, denoted by $x \sqcap y$. The binary operators \sqcup and \sqcap are respectively called the *join* and the *meet* of \mathcal{L} .

A lattice can also be defined as an algebraic structure. Both definitions 12 and 13 are equivalent.

Definition 13 (Algebraic lattice). A lattice is a set \mathcal{L} with two commutative and associative binary operations \sqcup and \sqcap such that

$$\forall (x, y) \in \mathcal{L}^2, x \sqcap (x \sqcup y) = x = x \sqcup (x \sqcap y)$$

We can define a partial order \sqsubseteq over such a structure as:

$$\forall (x, y) \in \mathcal{L}^2, x \sqsubseteq y \Leftrightarrow x \sqcup y = y$$

And for this partial order, $x \sqcup y$ is the greatest lower bound of $\{x, y\}$, and $x \sqcap y$ is its least upper bound. Conversely, the join and meet of definition 12 satisfy the properties of definition 13.

Definition 14 (Semilattice). A join-semilattice is a partially ordered set where any pair has a least upper bound (called join). A meet-semilattice is a partially ordered set where any pair has a greatest lower bound (called meet).

Definition 15 (Bounded lattice). A bounded lattice is a lattice $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap)$ that has a greatest element \top , called top, and a least element \perp , called bottom. For the algebraic structure, this is equivalent to:

$$\forall x \in \mathcal{L}, x \sqcup \perp = x \wedge x \sqcap \top = x$$

Definition 16 (Complete lattice). A complete lattice is a partially ordered set in which every subset S has a least upper bound, denoted $\sqcup S$. This is equivalent to state that every subset S has a greatest lower bound, denoted $\sqcap S$.

A complete lattice \mathcal{L} is bounded: $\perp = \sqcap \emptyset$ and $\top = \sqcup \mathcal{L}$.

We have defined the basic notions of lattice structure as a mathematical object. However, we must not lose sight that the lattices used in abstract interpretation are composed of abstractions of a concrete semantics, and that a concretization function gives meaning to these abstractions by relating them to sets of concrete elements. The lattice structure of abstractions must be consistent with their concretization.

$$\begin{array}{ll}
\gamma : \mathbb{X}^\# \rightarrow \mathcal{P}(\overline{\mathbb{X}}) & x_1 \sqsubseteq x_2 \Rightarrow \gamma(x_1) \subseteq \gamma(x_2) \\
\gamma(\top) = \overline{\mathbb{X}} & \gamma(x_1) \cup \gamma(x_2) \subseteq \gamma(x_1 \sqcup x_2) \\
\gamma(\perp) = \emptyset & \gamma(x_1) \cap \gamma(x_2) \subseteq \gamma(x_1 \sqcap x_2)
\end{array}$$

Figure 2.4: Lattices

Definition 17. Let $(\mathbb{X}^\#, \gamma)$ be a set of abstractions of concrete elements in $\overline{\mathbb{X}}$. The lattice structure $(\mathbb{X}^\#, \sqsubseteq, \sqcup, \sqcap, \top, \perp)$ of $\mathbb{X}^\#$ must satisfy the properties stated in Figure 2.4:

- the partial order over abstractions entails the inclusion of their concretizations;
- the concretization of \top is the set of all concrete elements; the concretization of \perp is the empty set;
- the join is an over-approximation of the union of concrete sets;
- the meet is an over-approximation of the intersection of concrete sets.

2.2.4 Fixpoint Computation

Resolving the system of equations 2.3 can be reduced to the computation of a function fixpoint. Let (X, \sqsubseteq) be a partially ordered set, and $f : X \rightarrow X$ a function. A *fixpoint* of f is an element $x \in X$ such that $f(x) = x$. A *pre-fixpoint* of f (respectively a *post-fixpoint*) is an element $x \in X$ such that $x \sqsubseteq f(x)$ (respectively $f(x) \sqsubseteq x$). If it exists, the least fixpoint of a function f is denoted $\text{lfp}(f)$, and its greatest fixpoint is denoted $\text{gfp}(f)$. An important result for the abstract interpretation of programs is the fundamental theorem of Knaster-Tarski [Tar55]:

Theorem 3 (Knaster-Tarski fixpoint theorem). *Let \mathcal{L} be a complete lattice and let $f : \mathcal{L} \rightarrow \mathcal{L}$ be a monotonic function. Then the set of fixpoints of f is also a complete lattice. In particular, f has a least and a greatest fixpoint, and:*

$$\begin{aligned}
\text{lfp}(f) &= \sqcap \{x \in \mathcal{L} \mid x \sqsubseteq f(x)\} = \sqcap \{x \in \mathcal{L} \mid x = f(x)\} \\
\text{gfp}(f) &= \sqcup \{x \in \mathcal{L} \mid f(x) \sqsubseteq x\} = \sqcup \{x \in \mathcal{L} \mid x = f(x)\}
\end{aligned}$$

This theorem proves the existence of a solution of the equations 2.3 when the abstract domain is a complete lattice and the transfer functions are monotonic.

Let $G = (N, \text{init}, \text{final}, \Sigma, I, \Omega, T)$ be a program and let the abstract domain $(\mathbb{D}, \sqsubseteq, \sqcup, \sqcap, \top, \perp)$ be a complete lattice of abstractions of concrete states Σ . We write \mathbb{X} the set of functions $X : N \rightarrow \mathbb{D}$ from

program points to abstractions. We can lift the complete lattice structure of \mathbb{D} into \mathbb{X} :

$$\begin{aligned} X_1 \sqsubseteq_{\mathbb{X}} X_2 &\Leftrightarrow \forall n \in N, X_1(n) \sqsubseteq X_2(n) \\ X_1 \sqcup_{\mathbb{X}} X_2 &\triangleq \lambda n. X_1(n) \sqcup X_2(n) \\ X_1 \sqcap_{\mathbb{X}} X_2 &\triangleq \lambda n. X_1(n) \sqcap X_2(n) \\ \top_{\mathbb{X}} &\triangleq \lambda n. \top \\ \perp_{\mathbb{X}} &\triangleq \lambda n. \perp \end{aligned}$$

One can easily prove that $(\mathbb{X}, \sqsubseteq_{\mathbb{X}}, \sqcup_{\mathbb{X}}, \sqcap_{\mathbb{X}}, \top_{\mathbb{X}}, \perp_{\mathbb{X}})$ is a complete lattice. Let $I^{\#} \in \mathbb{D}$ such that $I \subseteq \gamma(I^{\#})$. We consider the function $\mathcal{F} : \mathbb{X} \rightarrow \mathbb{X}$ defined as follows:

$$\mathcal{F}(X) = \lambda n. \begin{cases} I & \text{if } n = \text{init} \\ \bigsqcup_{(m, \text{stmt}, n) \in T} (\llbracket \text{stmt} \rrbracket^{\#}(X(m))) & \text{otherwise} \end{cases}$$

If each abstract transfer function $\llbracket \text{stmt} \rrbracket^{\#}$ is monotone, then the function \mathcal{F} is also monotone. In this case, the Knaster-Tarski theorem applies, and \mathcal{F} has a least fixpoint, which is a solution of equations 2.3. In practice, any fixpoint and post-fixpoint of \mathcal{F} is a solution of equations 2.3, but the least fixpoint is clearly the most precise abstract semantics of G .

Many further works have been devoted to the actual and efficient computation of such fixpoints. The Kleene fixpoint theorem expresses the fixpoint of a continuous function f (that preserves the lower upper bounds of chains) as the supremum of the ascending chain $(f^n(\perp))_n$. Cousot and Cousot [CC79a] have proven a constructive version of the Tarski theorem by means of transfinite iteration sequences. Bourdoncle [Bou93] studies precise and efficient algorithms for computing approximate fixpoints through chaotic iteration strategies; it especially targets the systems of semantic equations used in abstract interpretation. In practice, these systems are solved by iterative dataflow analysis [NNH99].

This thesis does not tackle the problem of the efficient computation of fixpoints. It assumes provided a fixpoint engine able to solve the equations of theorem 2 if the abstract semantics has the right properties: the abstract domain is a complete lattice, and the transfer functions are monotonic. This thesis is dedicated to the design of such abstract semantics, and more specifically to the combination of several abstract semantics. This is why the graph representation of programs is really convenient: it abstracts the control-flow of programs, which is handled by the fixpoint engine through a dataflow analysis, and let us focus on the statement semantics.

2.2.5 Widening

We have seen that the abstract interpretation theory expresses an abstract semantics of programs as a system of equations (theorem 2). These systems are solved by applying iteratively the equations until reaching a post-fixpoint. However, these equations may require a large number of iterations to be solved. Especially when the abstract domain is infinite or simply disproportionate, the convergence may be extremely slow. Moreover, some abstract domains used in real-world analyzers do not have a complete lattice structure. To accelerate the convergence, abstract interpretation relies on an extrapolation operator called *widening* [CC92b; Coro8], usually denoted by ∇ , and provided by each domain. Precise widenings usually study the differences between the states computed by two successive iterations, and extrapolate the effect of the next iterations to predict a possible fixpoint. In the dataflow analysis of a program, this consists in generalizing the behaviors of a loop from the properties of the first iterations.

Definition 18 (Widening). Let $(\mathcal{L}, \sqsubseteq)$ be a partially ordered set. A widening operator is a function $\nabla : \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{L}$ such that:

- For all pairs of elements $x, y \in \mathcal{L}^2$, we have $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$.
- for all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \dots \sqsubseteq x_n \sqsubseteq \dots$, the increasing chain defined as:

$$\begin{aligned} y_0 &\triangleq x_0 \\ y_{n+1} &\triangleq y_n \nabla x_{n+1} \end{aligned} \tag{2.4}$$

is ultimately stationary, i.e. $\exists k \in \mathbb{N}, \forall j \geq k, y_j = y_k$.

The first property of the definition ensures the soundness of the operator, that acts like the join of two abstractions. The second property allows a fast convergence of the iterative resolving of equations. We consider the monotone function $\mathcal{F} : \mathbb{X} \rightarrow \mathbb{X}$ defined in Section 2.2.4, whose post-fixpoints are solutions of the equations 2.3. We now define the increasing chain y_n as:

$$\begin{aligned} y_0 &\triangleq \perp \\ y_{n+1} &\triangleq y_n \nabla \mathcal{F}(y_n) \end{aligned}$$

The chain (y_n) is increasing as $y_n \sqsubseteq y_n \nabla \mathcal{F}(y_n)$. By definition of the widening, this chain is stationary starting from a $k \in \mathbb{N}$. We then have:

$$\mathcal{F}(y_k) \sqsubseteq y_k \nabla \mathcal{F}(y_k) = y_{k+1} = y_k$$

and y_k is thus a post-fixpoint of \mathcal{F} .

While ensuring a fast convergence of the fixpoint computation, the extrapolations made by a widening operator may be very imprecise and lose crucial information, thus leading to a post-fixpoint much greater than the least fixpoint. Several techniques can be implemented to mitigate this loss of precision.

A worthwhile observation is that all iterations do not need to apply the widening operator. It can be safely replaced by the standard join of abstractions at some iterations, as long as the widening is applied often enough to guarantee the convergence. The join ensures that the chain (y_n) is increasing, and the widening ensures the chain $(y_{\phi(n)})$ is ultimately stationary, with $\phi(n)$ being the steps where the extrapolations are done. Above a certain point however, if ϕ is not the identity, the widening must be applied between the last abstraction and the last *widened* abstraction, such that $y_{\phi(n)} = y_{\phi(n-1)} \nabla y_{\phi(n)-1}$. How often to extrapolate the next iteration by widening is a trade-off between precision and efficiency. Different heuristics can be used to choose when to widen [Cou+09]. A common strategy is to delay the widening [Bla+07; Bag+05], replacing the extrapolation operator by the standard join of abstractions for the first n iterations. This allows the computed abstractions to accumulate more constraints that can be used for a better extrapolation of a loop behavior. Two extrapolation steps can also be separated by m iterations without widening.

A *widening with thresholds* [LJG11; Lea+08; Hal93] can also be used to limit the extrapolations made by standard widenings. It consists in defining a finite set C of threshold constraints that express candidate properties that may hold in the concrete semantics of the analyzed program, and that we enforce in the widening. The widening between y_n and y_{n+1} is then reduced by the constraints that are still satisfied in y_{n+1} . The constraints can be expressed as a set of threshold abstract states TH . The widening with thresholds is then built upon the standard widening as follows:

$$x \nabla_{TH} y = \min(z \in TH \cup \{x \nabla y\} \mid y \sqsubseteq z)$$

The widening with thresholds $x \nabla_{TH} y$ is the least threshold state greater than y , or the standard widening $x \nabla y$. The precision and the efficiency of this approach depends on the choice and the number of the thresholds.

Finally, the over-approximations made for the computation of a post-fixpoint can be corrected afterwards, by decreasing iteration sequence [Cou99]. Dually to the widening operator, a narrowing operator Δ can be defined to accelerate the decreasing iterations [CC92b].

2.2.6 Abstract Domains: Summary

The abstract interpretation theory approximates the collecting semantics of a program through an abstract semantics expressed as equa-

tions over an abstract domain. An abstract domain is a collection of abstract states, related to sets of concrete states through a concretization function γ . An abstract domain provides:

- an inclusion and a join operations that over-approximate the inclusion and the union of sets of concrete states, according to γ .
- a set of transfer functions, or abstract transformers, that over-approximate the concrete semantics of statements in the collecting semantics.

As shown by Section 2.2.2, these are the only requirements that an abstract domain must fulfill to ensure the soundness of an abstract interpretation based analysis. Guaranteeing the termination of the analysis also requires the abstract domain to have a complete lattice structure, and its transfer functions to be monotone. In practice however, the fast convergence of the analysis is usually achieved through the extrapolations made by a widening operator. This technique makes unnecessary the other restrictions on abstract domains. Most abstract interpreters use relaxed hypotheses on their abstract domains and rely only on widenings to limit their analysis time. Several domains proposed in the literature do not have a complete lattice structure. Gange et al. [Gan+13] explore the abstract interpretation over domains that do not have a lattice structure. The Astrée static analyzer includes domains without a lattice structure, without even a preorder, and whose transfer functions are not monotonic [Cou+06].

However, the lattice structure of abstractions brings good properties to an abstract interpretation based analysis, even if it is not strictly needed for its termination or its soundness. In particular, the commutativity and the associativity of the join and meet operators ensure that the iterations order does not affect the result (and thus the precision) of the analysis. If we do not enforce complete lattices or monotonic transfer functions, we assume henceforth that the abstractions we use have a bounded lattice structure —or at least a semilattice structure.

2.3 COMBINATION OF ABSTRACTIONS

Over the years, many abstract domains have been designed to address various problems on different classes of programs. The complete verification of a program often requires properties inferred by different domains. The abstract interpretation framework provides some standard ways to create new abstractions from existing ones, especially by combining abstract domains and their abstract semantics. Most of them allow abstract domains to interact with each other during the analysis, exchanging information in order to reach more precise abstractions of the program semantics. Cortesi et al. [CCF13] offers a good survey of the product operators on abstract domains in the

Domain	Reference	Properties
Signs	[CC77b]	$x \in \pm$
Intervals	[CC77b]	$x \in [a..b]$
Boxes	[GC10a]	$x \in \bigcup_{i=1}^n [a_i..b_i]$
Congruences	[Gra89; Gra97]	$x \equiv c \pmod{m}$
Linear congruences	[Gra91]	$\sum_{i=1}^n (a_i \cdot x_i) \equiv c \pmod{m}$
Linear equalities	[Kar76]	$\sum_{i=1}^n (a_i \cdot x_i) = n$
Polyhedra	[CH78]	$\sum_{i=1}^n (a_i \cdot x_i) \leq n$
Octagons	[Mino6c]	$\pm x \pm y \leq c$
Ellipsoids	[Fero4]	$ax^2 + by^2 + cxy \leq n$

$$a, b, c, n, m \in \mathbb{N}$$

$$x, y \in \mathcal{X}$$

Table 2.1: Numerical abstract domains in the literature

literature. In this section, we first give an overview of the existing abstract domains; we then present different approaches to combine abstract domains.

2.3.1 Abstract Domains of the Literature

A wide variety of abstract domains have already been proposed in the literature, and the design of new abstract domains remains a very active field of research. Each domain has its strengths and weaknesses. They provide different expressiveness and computational complexity. They infer different families of properties, handle precisely various kinds of code patterns and offer different balances between precision and efficiency. *Relational* domains encode properties that involve multiple variables. Some relational domains limit the number of variables that can be related through the properties they infer. *Non-relational* domain abstracts each variable independently.

Table 2.1 presents some of the well-known numerical abstract domains in the literature. The family of properties that each domain may infer is given to the right. An abstract state represents the set of concrete states in which the properties hold.

SIGNS The sign domain [CC77b] is one of the simplest abstractions known in abstract interpretation: it only infers the sign of each variable of the program.

INTERVALS The interval domain [CC77b] represents the range of variables by maintaining an upper bound and a lower bound for their possible values. It relies on the well-known interval arithmetics [Moo66]. The interval domain is very popular as it is able to infer crucial information for program verification despite its low complexity.

BOXES Boxes [GC10a] are a refinement of the interval domain: they represent the possible values of each variable as a disjunction of intervals, encoded by a propositional formula over interval constraints and represented by Linear Decision Diagrams, an extension of Binary Decision Diagrams.

CONGRUENCES The arithmetical congruence domain [Gra89; Gra97] infers congruence constraints for each variable independently. This domain is especially useful for variables used as array indices, that are most of the time congruent to the size of the array elements.

LINEAR EQUALITIES A linear equality domain [Kar76] detects affine relationships among program variables.

LINEAR CONGRUENCES A linear congruences domain [Gra91] discovers a system of linear congruence equations satisfied by the integer variables of a program. It generalizes the domain of arithmetical congruences into a fully relational domain.

POLYHEDRA A convex polyhedra domain [CH78] manipulates systems of linear inequalities between program variables. It can express complex invariants at the cost of a very high complexity.

ZONES & OCTAGONS Zones [Mino01] and octagons [Mino06] aim to find an happy medium between the expressiveness and the cost of the polyhedron and the interval domains. They use Different-Bound Matrices [Yov96; Lar+97] to efficiently encode bounds on the sum and the difference of pairs of variables.

ELLIPSOIDS Ellipsoids [Fero04] are a specialized domain dedicated to the analysis of digital filters.

These are only numerical domains on arithmetical variables, but the abstract interpretation framework has also been successfully applied to prove various properties on memory. Abstract domains have been notably designed to infer information about pointer aliasing [Hino1; RL12], large or complex memory structures [VBo4; Mino6a] or heap-allocated storage [BR06; SRW02; TCR13].

2.3.2 Direct Product

We now study the composition of abstract domains within the abstract interpretation framework. Henceforth, we assume given two abstract domains $A^\#$ and $B^\#$ and their respective concretizations, join-semilattice structures with widenings and transfer functions.

$$\begin{array}{lll} \gamma_A : A \rightarrow \mathcal{P}(\Sigma) & (A^\#, \sqsubseteq_A, \sqcup_A, \nabla_A) & \llbracket \text{stmt} \rrbracket_{A^\#} : A \rightarrow A \\ \gamma_B : B \rightarrow \mathcal{P}(\Sigma) & (B^\#, \sqsubseteq_B, \sqcup_B, \nabla_B) & \llbracket \text{stmt} \rrbracket_{B^\#} : B \rightarrow B \end{array}$$

The direct product of two abstract domains is simply the set of pairs of elements of each domain. A pair (a, b) from $A \times B$ abstracts the concrete states represented by both abstract states a and b . The operators are defined by the pointwise application of the operator of each underlying domain. This trivially preserves the soundness properties of the domains, as well as their (possibly complete) lattice structure, the monotony of the transfer functions and the definition 18 of widening.

$$\begin{aligned} \gamma_{A \times B} : A \times B &\rightarrow \mathcal{P}(\Sigma) \\ \gamma_{A \times B}(a, b) &\triangleq \gamma_A(a) \cap \gamma_B(b) \\ (a, b) \sqsubseteq_{A \times B} (a', b') &\Leftrightarrow a \sqsubseteq_A a' \wedge b \sqsubseteq_B b' \\ (a, b) \sqcup_{A \times B} (a', b') &\triangleq (a \sqcup_A a', b \sqcup_B b') \\ (a, b) \nabla_{A \times B} (a', b') &\triangleq (a \nabla_A a', b \nabla_B b') \\ \llbracket \text{stmt} \rrbracket_{A \times B}^\#(a, b) &\triangleq (\llbracket \text{stmt} \rrbracket_{A^\#}(a), \llbracket \text{stmt} \rrbracket_{B^\#}(b)) \end{aligned}$$

The direct product allows an automatic parallelization of analyzes based on different abstract domains, but it is not more precise than the sequence of these analyzes. It discovers in one analysis the invariants inferred by its underlying domains in separate analyzes. While the direct product adds no intrinsic complexity to the abstract semantics of the domains, more accurate analyzes can be obtained by mutually refining the underlying abstract domains.

Example 2. Let us consider the program of Figure 2.5 that operates on variables i , x and y . The assumption of line 2 constrains the set of initial states of the program: they are environments in which the variable i has a value between 0 and 3. Such an assumption can also be translated as a test filter of the toy language, as in the graph representation given at the right of the figure. In this case, the program accepts any initial state, but those that do not satisfy the condition are stuck in the initial node. Both representations are equivalent.

On this program, the interval domain infers that $i \in [0..3]$ initially, that $x \in [0..12]$ at line 1, that $x \in [1..12]$ at line 3 and finally that $x - i \in [-2..12]$. This does not exclude the division by zero at line 3. The arithmetic congruence domain infers that $x \equiv 0 \pmod{4}$, but has

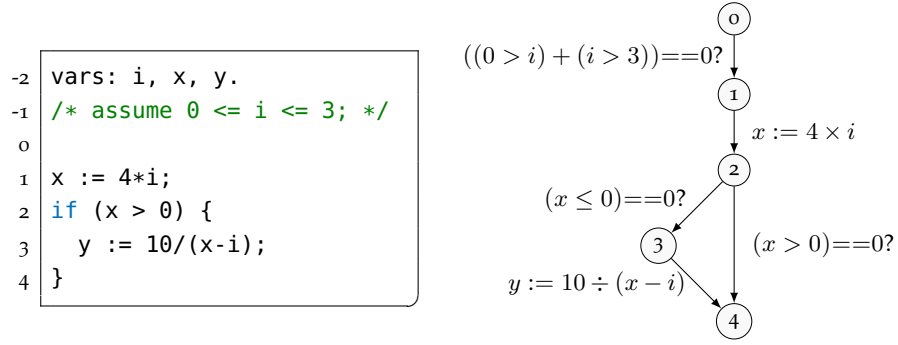


Figure 2.5: Need for interactions between abstract domains

no information about the expression $x - i$ at line 3. Therefore, the direct product between an interval domain and an arithmetic cannot prove that the division by zero never occurs on this program. Nevertheless, the product of abstract states inferred at line 3 contains all information needed to prove this property:

$$\begin{array}{l|l}
 i \rightarrow [0..3] & \\
 x \rightarrow [1..12] & x \equiv 0 \pmod{4} \\
 y \rightarrow \top &
 \end{array} \quad (2.5)$$

The two constraints on variable x ensure that the value of x belongs to the interval $[4..12]$, and thus that the expression $x - i$ belongs to the interval $[1..12]$, excluding the division by zero. Here, proving the correctness of the program requires an inter-reduction between the properties inferred by each abstract domain. This is exactly the purpose of a *reduced product*.

2.3.3 Reduced Product

The reduced product [CC79b] is a refinement of the direct product that overcomes its limitations by mutually refining the underlying abstract states. Intuitively, it reduces the abstract state of each domain by exploiting the properties inferred by the other. These reductions improve the information known by each component without affecting the meaning (the concretization) of the pair. This allows each domain to approximate more precisely the program without compromising the soundness of the analysis. Formally, the reduced product considers the equivalence classes of the direct products that have the same concretization. Practically, it works on the smallest representatives of each class, by reducing maximally the result of each operation through a reduction operator. $\text{reduce}(a, b)$ computes the smallest element of $A \times B$ that has the same concretization as (a, b) . If both

underlying domains have a complete lattice structure, it can be defined as the meet of the equivalence class of (a, b) .

$$\begin{aligned} \text{reduce}(a, b) &= \min_{\sqsubseteq_{A \times B}} ((a', b') \mid \gamma_{A \times B}(a', b') = \gamma_{A \times B}(a, b)) \\ &= \bigcap \{(a', b') \mid \gamma_{A \times B}(a', b') = \gamma_{A \times B}(a, b)\} \end{aligned}$$

The reduced product $A * B$ is then defined as the set $A \times B$ with the following operators. The concretization is the same as for the direct product. The lattice operators and the transfer functions use the reduction operator to minimize their output in the direct product. As its concretization remains unchanged, this does not affect the soundness of the product domain. On the other hand, the result of the widening cannot be reduced without breaking the properties stated in definition 18. A widened state is thus not necessarily the smallest element of its equivalence class. This is why the reduced product cannot be defined as $\{\text{reduce}(a, b) \mid (a, b) \in A \times B\}$.

$$\begin{aligned} \gamma_{A*B}(a, b) &\triangleq \gamma_A(a) \cap \gamma_B(b) \\ (a, b) \sqsubseteq_{A*B} (a', b') &\Leftrightarrow a \sqsubseteq_A a' \wedge b \sqsubseteq_B b' \\ (a, b) \sqcup_{A*B} (a', b') &\triangleq \text{reduce}(a \sqcup_A a', b \sqcup_B b') \\ \llbracket \text{stmt} \rrbracket_{A*B}^\#(a, b) &\triangleq \text{reduce}(\llbracket \text{stmt} \rrbracket_A^\#(a), \llbracket \text{stmt} \rrbracket_B^\#(b)) \\ (a, b) \nabla_{A*B} (a', b') &\triangleq (a \nabla_A a', b \nabla_B b') \end{aligned}$$

Example 3. On the program of Figure 2.5, the direct product state inferred at line 3 was given by equation 2.5. Its reduction in the reduced product is:

$$\left. \begin{array}{l} i \rightarrow [0..3] \\ x \rightarrow [4..12] \\ y \rightarrow \top \end{array} \right| x \equiv 0 \pmod{4}$$

The interval component is then able to prove the absence of a division by zero in the program.

Although the mathematical definition of the reduced product enjoys ideal theoretical properties, the maximal reduction between arbitrary domains is not computable in general. The implementations of domains composition usually fall back to an approximate reduced product, through a relaxed version of the reduction operator. Such a relaxed operator partially reduces a pair of abstract states, satisfying the following properties:

$$\begin{aligned} &\text{reduce} : A \times B \rightarrow A \times B \\ \forall (a, b) \in A \times B, &\quad \begin{cases} \text{reduce}(a, b) \sqsubseteq_{A \times B} (a, b) \\ \gamma_{A \times B}(\text{reduce}(a, b)) = \gamma_{A \times B}(a, b) \end{cases} \end{aligned}$$

The choice of the reductions performed within an approximate reduced product must reach a balance between precision and efficiency. A standard (and trivial) reduction is the normalization of a canonical bottom element: any pair (\perp_A, b) or (a, \perp_B) has an empty concretization and can be replaced by (\perp_A, \perp_B) . The specification and the complexity of the stronger reductions depend entirely on the involved abstract domains.

2.3.3.1 Granger Product

Granger [Gra92] proposes a convenient approach to compute an approximation of the reduce function. It simplifies its implementation by separating the reduction of each component of the pair. The reductions are split up into two operations $\varrho_A : A \times B \rightarrow A$ and $\varrho_B : A \times B \rightarrow B$ such that:

$$\forall (a, b) \in A \times B, \begin{cases} \gamma(a, b) = \gamma(\varrho_A(a, b), b) = \gamma(a, \varrho_B(a, b)) \\ \varrho_A(a, b) \sqsubseteq_A a \wedge \varrho_B(a, b) \sqsubseteq_B b \end{cases}$$

Each operator refines the state of a domain with the information of the other state. Both operators can then be used iteratively until reaching a fixpoint. The Granger product of a pair (a, b) is defined as the fixpoint of the decreasing sequence:

$$\begin{aligned} (a_0, b_0) &\triangleq (a, b) \\ (a_{n+1}, b_{n+1}) &\triangleq (\varrho_A(a_n, b_n), \varrho_B(a_n, b_n)) \end{aligned}$$

2.3.3.2 Limitations of the Reduced Product

Approximate reduced products are well suited on some specific abstract domains, as proved by the well-known product of interval and congruence domains [Gra89]. Nevertheless, the reduced product is not the panacea of domain composition. Two main impediments hinder the spread of its use for arbitrary abstract domains.

1. Reduced products are not modular: the reduction operator is specific to the domains it refines. It can be hard to define on rich domains, and is even more intricate to apply on a high number of domains. Adding a new abstract domain to a Granger product of n domains may require the implementation of $2n$ reduction functions, and each one may involve different alternatives to approximate the maximal reduced product.
2. The reduced product performs the inter-reduction of domains on the result of the transfer functions. There is no interaction between abstract domains *during* their interpretation of the statement semantics. However, an abstract domain could exploit the properties inferred by other domains to approximate more accurately the semantics of a statement. The shared properties could

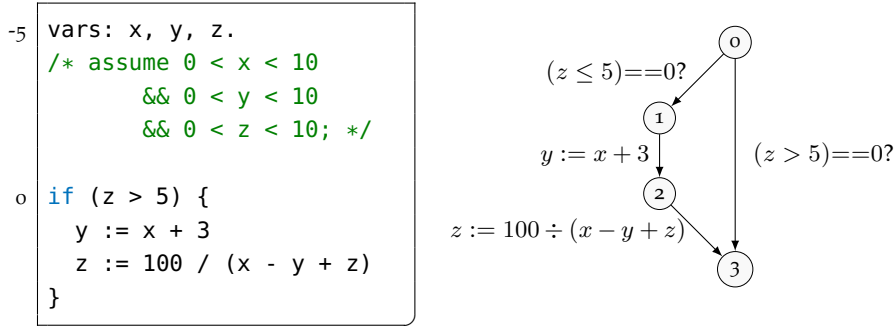


Figure 2.6: Need for interactions during statements interpretation

be lost or pointless after the statement, preventing the reduced product from recovering the same precision.

Example 4. The program of Figure 2.6 operates on three variables x , y , z . The assumption constrains the initial states. We study the analysis with the domains of intervals and linear equalities. An interval analysis infers the initial range $[1..9]$ of each variable, and the properties $z \in [6..9]$ at line 1 and $y \in [4..12]$ at line 2. The interval arithmetics leads to $x - y + z \in [-6..11]$ at line 2. The linear equalities domain cannot express a constraint over z , and only infers $y = x + 3$ at line 2. None of the two domains can exclude the possibility of a division by zero. The states of each domain cannot be mutually refined here: their product is already minimal. The state computed by the reduced product at line 2 is:

$$\begin{array}{l}
 x \in [1..9] \\
 y \in [6..12] \quad \times \quad y = x + 3 \\
 z \in [6..9]
 \end{array}$$

Therefore, the reduced product of intervals and linear equalities cannot prove the correctness of the program. However, the equality $y = x + 3$ implies that $x - y \in [-3]$. As the interval domain ensures that $z \in [6..9]$, this proves that $x - y + z \in [3..6]$. The program can thus be proved correct by a combination of the interval and the equality domain, provided that both domains can collaborate during the approximation of a statement semantics.

Example 5. A more classic example of this limitation of reduced products is the abstract interpretation of programming languages including arrays. A precise approximation of an array access $t[i]$ often requires at the same time some bounds *and* a congruence information about the possible values of i . This requires some interaction between an interval and a congruence domain within the transfer functions. Otherwise, the loss of precision can be too big to be recovered afterwards, especially on large arrays.

2.3.4 Open Product

The open product [CCH00] is a framework to easily implement interactions between abstract domains during their abstract operations. It bases their inter-reductions on *queries* and *open operations*. A query is a boolean function that expresses a property inferred by an abstract state. An open operation is a transfer function that additionally receives a set of queries describing some properties that are known to hold. The transfer functions over an abstract domain can use the queries provided by other domains to improve their precision. Information thus flows between abstract domains through queries. In this context, an open interpretation is an abstract domain that implements a set of queries Q , and open operations with respect to the same set of queries Q . The open product of several open interpretations over the same queries Q is defined as:

- the cartesian product of domains;
- queries as the disjunction of the queries of each domain (as $\text{true} \leq \text{false}$ is used as the order on booleans, $\lambda x. q_A(x) \vee q_B(x)$ is the most precise query logically implied by q_A and q_B);
- the component-wise application of the open operations on each domain, with the queries defined above.

While the open product has been specifically designed to let domains interact during a statement interpretation, a reduce operator can also be used to refine the result of the open operations. In particular, the open product is orthogonal and complementary to the Granger product.

The main drawback of the open product is the restriction to boolean functions, that lack expressivity. Moreover, designing an open product requires to properly identify beforehand the set of properties that need to be exchanged by abstract domains. Finally, this communication system could be extended to improve the join of abstract states: queries are only used by open operations, but could also be used at join points to mitigate their usual loss of precision.

2.3.5 Communication through a Shared Language

We have seen that the reduced product is highly non modular and thus cumbersome to apply on a wide variety of domains. A natural solution to this issue is the use of an independant language of constraints \mathcal{L} as intermediary between abstract domains [Cou+06]. Such a language is for instance defined in [TCR13] for the inter-reduction between shape abstract domains. Following the Granger product idea, one can interface an abstract domain \mathbb{D} with the language \mathcal{L} by extending it with two functions:

- $extract : \mathbb{D} \rightarrow \mathcal{L}$, that expresses the constraints guaranteed to hold by an abstract state;
- $refine : \mathbb{D} \times \mathcal{L} \rightarrow \mathbb{D}$, that refines an abstract state according to some constraints.

If we assume the constraints \mathcal{L} equipped with a concretization function $\gamma_{\mathcal{L}} : \mathcal{L} \rightarrow \mathcal{P}(\Sigma)$, the two functions must satisfy the following properties (the two first are soundness properties, and the third states that the refinement of an abstract state d cannot be less precise than d):

$$\begin{aligned} \gamma(d) &\subseteq \gamma_{\mathcal{L}}(extract(d)) \\ \gamma(d) \cap \gamma_{\mathcal{L}}(l) &\subseteq \gamma_{\mathcal{L}}(refine(d, l)) \\ refine(d, l) &\sqsubseteq d \end{aligned}$$

An approximate reduced product can then be implemented by successive iterations of extracting constraints from states and refining states with these constraints. The constraint language should be designed in advance with utmost care, as it is the fixed interface between all abstract domains. Unlike the open product, this extension of domains does not allow interactions during the interpretation of statements.

2.3.6 Communication by Messages

The most advanced collaboration between abstract domains is unquestionably the communication by messages introduced and implemented in Astrée [Cou+06]. This design has been followed later in other analyzers such as Canal [Dup13] and Verasco [Jou+15]. Its main idea is to use as shared interface between domains an extensible set of atomic messages. The messages can be of various kinds; each one carries a different type of information. For instance, Verasco [Jou16] uses different messages to express the interval of a variable, the arithmetical congruence of a variable, the exact value of a variable, the equality between a variable and an expression, or the equality between an expression and a quasi-linear expression. Communication channels relay a list of messages from domains to domains. Each domain can use some of the messages from a channel, and emit some new messages of its own. A domain is always free to ignore the messages it does not understand. This allows adding a new kind of message without having to modify all the existing domains.

A domain creates messages and is reduced according to messages through the two functions *extract* and *refine* presented in the above subsection. Following the concepts of the open product, the transfer functions can also exploit the information conveyed by channels. Several channels can enable various forms of communication between abstract domains. Astrée uses channels of two types:

- input channel for a domain to request more information. An input channel is implemented as a list of functions, each one computing messages of a fixed kind.
- output channel for a domain to emit a message intended for other domains. An output channel can be implemented as a list of messages to be processed.

2.3.6.1 Input Channels

An input channel works as the set of queries in the open product. The communication model of Astrée actually extends the open product in two different ways:

- the functions of an input channel produce messages and not simply booleans;
- two input channels are provided: a *pre* and a *post* channels, that contain messages describing properties known to hold respectively before and after the given statement.

The new signature of transfer functions is given below. The transfer function $\{\text{stmt}\}^\#((d, pre), post)$ over-approximates the semantics of *stmt* on the concrete states represented by *d* and satisfying *pre*, knowing that the resulting states satisfy *post*.

$$\begin{aligned} \{\text{stmt}\}^\# : (\mathbb{D} \times \mathcal{L}) &\rightarrow \mathcal{L} \rightarrow \mathbb{D} \\ \{\text{stmt}\}(\gamma(d) \cap \gamma(pre)) \cap \gamma(post) &\subseteq \gamma(\{\text{stmt}\}^\#((d, pre), post)) \end{aligned}$$

The *pre* channel is collaboratively built by the product of abstract states inferred at the program point before the statement —this states product is the argument of the transfer function. The *post* channel is built from the abstract states inferred after the statement —this states product is the result of the transfer function. The *post* channel is thus incrementally filled by each abstract state computed by the transfer function of one of the domains. The evaluation order of abstract domains thereby affects the communication between them. The first domain to be processed has always an empty *post* channel. The n^{th} domain to be processed benefits from the messages created by the previous domains, and can add some new messages for the next ones.

2.3.6.2 Output Channels

While an input channel allows a domain to request information, an output channel allows a domain to send information on its own initiative. Input and output channels are used in parallel by the transfer functions. Astrée uses two output channels:

- an oriented output channel, that allows a domain to send a message to the next domains to be processed;

- a broadcast output channel, that allows a domain to send a message to all domains (including those that have already performed their computation).

At the beginning of the interpretation of a statement, both output channels are empty. Messages are incrementally provided by each domain when interpreting the statement (through the transfer functions). The contents of the oriented channel can directly be used by the transfer function of a domain to be more accurate. The use of the broadcast channel is postponed to the end of the interpretation of the statement. The content of the broadcast channel is only then sent to all domains, that finally have the opportunity to use its information.

2.3.6.3 *Summary*

The communication by messages resolves the two main issues of the reduced product. It is modular, as a domain can be freely added or removed from a product without directly impacting the others. It is extensible, as adding a new kind of message does not require the communication system or all existing domains to be modified. Finally, it enables various powerful interactions between abstract domains. However, this extensive leeway for domain collaboration comes at the price of some complexity. The communication system can be invasive in the implementation of the abstract interpreter. It has to be maintained in parallel of the abstract semantics implemented by the domains. It features four different channels of two different types that must be handled by the abstract domains and their transfer functions. The product of domains is ordered, and the order impacts the possible interactions. Moreover, each domain needs to choose carefully how to exchange each atomic information, as each channel reaches different domains at different costs.

2.3.7 *Abstract Interpretation Based Analyzers*

Over the years, the fundamental principles of the abstract interpretation theory have been applied to develop static analyzers, which are software that aim at proving automatically the absence of some undesirable behaviors in real-world programs. The undesirable behaviors tracked by an analyzer are generally the runtime errors of a programming language, modeled by the error states Ω in our concrete semantics. Static analysis has already shown its industrial applicability to prove safety properties on critical or embedded code, and abstract interpretation has known considerable progress in recent years, in terms of both research breakthrough and industrial-strength implementations. However, although most abstract interpreters use multiple domains internally, few explain how the different domains

exchange information. We present here some of the worthwhile tools based on abstract interpretation.

IKOS [Bra+14] is an open-source framework that supports the development of static analysis based on abstract interpretation. It provides a front-end for C/C++ programs based on LLVM [LA04], some optimized fixpoint algorithms and several numerical abstract domains including octagons, discrete symbolic domains, and a reduced product between intervals and congruences. It is used for instance by the verification software SeaHorn [Gur+15].

APRON [JMo9] is an open-source library providing numerical abstract domains of the literature under a unified interface. It currently contains an implementation of intervals, octagons, linear equalities, convex polyhedra, linear congruences, and a reduced product between polyhedra and linear congruences. The Interproc analyzer [Ja] has been designed to demonstrate the features of the APRON library.

ASTRÉE [Bla+02; Bla+07; Cou+05; Cou+09] is an abstract interpreter tailored towards the verification of safety-critical embedded real-time softwares written in C. It contains many numerical abstract domains able to handle heterogeneous kinds of abstract properties and to prove complex numerical invariants. Astrée enjoys a modular architecture that organizes a hierarchy of domains, and features the advanced communication by messages between them described above. It has been successfully applied to prove the absence of runtime errors in the flight control codes of the Airbus fly-by-wire systems.

VERASCO [Jou+15] is an abstract interpreter that is itself formally proved using the Coq proof assistant. Verasco is integrated into CompCert [Ler09], a formally-verified compiler for C, in such a way that the safety properties established by Verasco carry over the compiled code. The design of Verasco follows closely that of Astrée, especially for the interactions between abstract domains.

CLOUSOT [FL10] checks the absence of runtime errors, but also verifies the contract specifications of functions. It analyzes each function of a program in isolation, using its pre- and post-conditions. It works at the bytecode level and is thus language independent. It organizes several domains in a hierarchy inspired by that of Astrée, and enables some communications between them. Clousot relies on an abstract interpretation engine, but also embodies other techniques such as goal-directed backward propagations or the inference of pre- and post-conditions.

FRAMA-C / VALUE FRAMA-C [Kir+15] is an extensible and collaborative platform dedicated to the specification and the verification

of C programs. For this purpose, Frama-C provides several plugins based on different techniques. Among them, the [VALUE](#) plugin uses abstract interpretation to compute a variation domain for the variables of a program. Unlike other analyzers, the [VALUE](#) architecture is not modular: it contains only one abstract domain and does not allow the introduction of new abstractions. The work presented in the next chapter and developed in the remaining of this thesis extends the [VALUE](#) plugin to overcome this limitation. The FRAMA-C platform is introduced in more details in Section [3.1.1](#), and the differences between the former [VALUE](#) analyzer and our work are emphasized in Section [3.4](#).

Part II

EVA: A MODULAR ANALYZER FOR C

ARCHITECTURE OF THE ANALYZER

This chapter outlines the main ideas that have guided the development of [EVA](#), and that still underlie its architecture. Each of the upcoming chapters focuses on a specific component of [EVA](#), while this chapter explains how these components fit together and interact with each other to form a complete abstract interpreter. Thus, it does not dwell on the details or the formalization of the analyzer, but introduces its main features.

[EVA](#) being a plugin of the FRAMA-C platform, we start by briefly presenting this framework. We then describe the modular architecture of [EVA](#), which relies on a particular hierarchy of different abstractions. We explain the role of these abstractions in the generic abstract interpreter, and sketch the communication between them. We also propose an implementation mechanism that enables some interaction between the components of a generic combination of datatypes. This has been successfully implemented to improve the product of abstractions in [EVA](#). We finally discuss the software development of the analyzer, which is based on [VALUE](#), the previous abstract interpreter of FRAMA-C.

3.1 OVERVIEW OF THE EVA STRUCTURE

Figure [3.1](#) sketches the different layers of our analyzer architecture. The services each layer provides are given on the left, and the syntax fragments on which they operate are on the right. This section follows the internal organization of the analyzer. We first introduce FRAMA-C, the platform that [EVA](#) is part of. Then, we describe the abstract interpreter itself, and the hierarchy of abstractions it uses to represent program semantics.

3.1.1 *The Frama-C Platform*

The [EVA](#) abstract interpreter is a plugin of the FRAMA-C [[Kir+15](#)] platform: an extensible and collaborative framework dedicated to the analysis of the source code of C programs. It is structured around a kernel providing shared functionalities, and a set of plugins that implement various types of verification. The plugins may interact with each other, and a plugin may build upon the properties proved by another analysis. In particular, the results inferred by [EVA](#) are available to the other plugins of FRAMA-C.

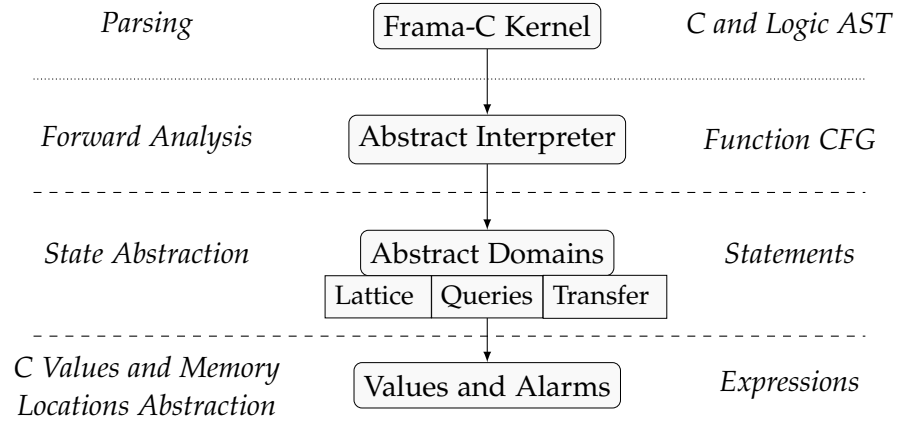


Figure 3.1: Overall layers of EVA

Available analyses in FRAMA-C include abstract interpretation (EVA plugin), deductive verification (WP plugin), generation of temporal annotations (aorai plugin), runtime assertion checking (E-ACSL plugin), etc. A more exhaustive list can be found at <http://frama-c.com/plugins.html>. Note that EVA is not a new plugin of FRAMA-C, but a major evolution of its former abstract interpreter, the Value Analysis plugin (often abbreviated as VALUE).

Among the main features provided by the kernel are:

- the parsing and the type-checking of the C files supplied as input. A symbolic linker then produces a program equivalent to the input files. The resulting AST is expressed in the augmented FRAMA-C version of the C Intermediate Language [Nec+02].
- the possibility to specify C functions, using a first-order logic called ANSI/ISO C Specification Language (ACSL). Logical annotations may range from basic assertions (e.g. expressing that an operation does not overflow), to full-fledged functional specification of the function behavior.
- the tracking and automatic consolidation of the logical statuses (true, false or unknown) set by the plugins on the ACSL annotations present in the program [CS12].

The verification of a program generally starts by proving that it does not contain *runtime errors*, i.e. undefined behaviors according to the C standard (detailed in Section 4.1). This is often a prerequisite to more complex verifications such as functional correctness. Within FRAMA-C, safety assertions (that ensure the absence of undefined behaviors) can be generated in a systematic way by two plugins: RTE and EVA. Afterwards, all plugins may attempt to prove these assertions. EVA is somewhat special, in that it intermingles the generation of safety assertions and their verification. EVA is also able to emit logical statuses on simple user-written ACSL assertions.

FRAMA-C —and thus EVA— is written in OCaml [Cuo+09], and is an open source, free software. Its source code is available at <http://frama-c.com/download.html>. The first version of EVA was publicly released in FRAMA-C Aluminium in June 2016. This thesis is based on this work.

3.1.2 The Abstract Interpreter

The generic abstract interpreter works on the AST provided by the FRAMA-C kernel. It iterates an interprocedural forward dataflow analysis over the CFG of the functions of a given program. The analysis propagates the states of an *abstract domain* through the CFG, according to an abstract semantics soundly modeling the effect of statements. Statements are visited following a chaotic iteration strategy [Bou93]. Widening steps are performed to ensure convergence. The analysis of a whole program is fully context-sensitive: function calls are symbolically inlined. Recursive functions are currently not handled.

Once a fixpoint has been reached, the computed abstract states infer valid properties at each program point. During the analysis, EVA emits an *alarm* for each operation where the abstract semantics fails to prove the absence of an undesirable behavior (i.e. an undefined behavior that EVA tracks). Alarms are expressed as ACSL assertions that may be proved later by another plugin of FRAMA-C. Each alarm may report a real error if the reported undesirable behavior indeed occurs at runtime for at least one possible execution. Otherwise, the alarm is a false alarm, due to the over-approximation made by the abstract semantics. Importantly, the soundness of the abstract semantics ensures that an alarm is issued for each operation that may fail. If the analysis raises no alarm for a given program, then this program is free of the undesirable behavior that EVA detects.

The abstract interpreter is generic: it does not depend on a particular abstraction. The architecture and the inner working of this interpreter are further developed in Section 3.2.1. But before delving into the details of the interpreter, we need to present the hierarchy of abstractions on which it relies.

3.1.3 Abstractions

The design of EVA relies on the separation between *state abstractions* and *value abstractions*. As the name implies, a state abstraction represents some concrete states at a point of a program execution. A value abstraction represents the C values an expression can have in some concrete states. A state abstraction handles the semantics of statements, while a value abstraction operates at the level of expressions. This distinction between value and state abstractions was already proposed by Cousot [Cou99] to design an abstract interpreter, but was

not used to structure or reduce a product of different abstractions. On the other hand, *EVA* uses the value abstractions as a shared communication interface that allows different state abstractions to interact with each other.

Value abstractions and state abstractions are fully formalized in Chapter 5 and in Chapter 7 respectively. The main value abstraction used by *EVA* is described in Section 5.3. The state abstractions currently available in *EVA* are listed in Chapter 8. The following is an outline of the role of both kinds of abstractions in the abstract interpreter.

3.1.3.1 State Abstractions

An *abstract domain* —or a state abstraction— is a collection of *abstract states* which are abstractions of the sets of possible concrete states that may occur at a program point. An abstract domain must provide:

- a join-semilattice structure, fulfilling the properties established in Figure 2.4 (on page 30). The inclusion relation reflects the precision of the abstract states. The join computes an upper bound of two states, and is used when two branches of the CFG merge. A widening operator is also mandatory, to ensure a fast convergence of the fixpoint computation.
- sound transformers —or transfer functions— for statements, that infer properties by modeling the effect of a statement on an abstract state. They are the abstract semantics of the domain, and are used to propagate abstract states through the CFG. They must satisfy the property of definition 8 (on page 27).
- queries, which extract information from abstract states by assigning an abstract *value* to some expressions (including expressions denoting addresses). They are detailed in Chapter 7.

The abstract domains do not interact directly with each other, but achieve some communications and inter-reductions through value and locations abstractions expressed by the queries.

3.1.3.2 Value and Locations Abstractions

Value and location abstractions are non-relational abstractions about expressions and addresses in some given concrete states. A value abstraction is an approximation of the set of possible concrete values that an expression may have in these states. As expressions include dereferences of a program variable, a value abstraction may be an approximation of the concrete value of a variable in some states. A location abstraction is an approximation of the set of possible memory locations that an address may have. Both abstractions serve similar purposes, and must fulfill the same requirements. Although they are

internally two different entities, the value and location abstractions in [EVA](#) are very close, and share a large part of their implementation. Therefore, we usually do not distinguish between value and location abstractions, and often write “value abstraction” for both kinds of abstractions.

Value (and location) abstractions are used to cooperatively evaluate addresses and expressions in a product of abstract states coming from several abstract domains. Value (respectively location) abstractions encode an abstract semantics of the operators on expressions (respectively addresses). According to the properties it has inferred, an abstract state may express the possible concrete value of a variable, expression or address by providing such abstractions. These abstractions are then made available to the transformers of all domains, which can use them to approximate more precisely the effect of a statement.

Value and location abstractions must have a meet-semilattice structure. Similarly to state abstractions, the order relation reflects the precision of the abstractions. The meet is used to combine abstractions provided by different domains. Value and location abstractions also provide sound transformers over-approximating the effect of arithmetic operators on expressions and addresses. As the concrete operators may cause undefined behavior at execution time, their abstract counterparts also produce *alarms*, which report the error cases.

Alarms are essential: they are the final result of an analysis, and highlight all the possible undesirable behaviors that may happen during the execution of a given program. In return, the absence of alarm guarantees the absence of undesirable behavior. An abstract interpreter aims at emitting as few alarms as possible —while remaining sound. It is thus natural to also involve the state abstraction in their computation, allowing them to avoid some alarms thanks to their inferred properties.

Values, locations and alarms form the complete communication interface between the abstract domains and the generic abstract interpreter. They are formally described in [Chapter 5](#).

3.2 A MODULAR ABSTRACT INTERPRETER

This section offers a more detailed view of the inner workings of an abstract interpreter based on the separation between value and state abstractions. We also briefly explain the combination of abstractions, as well as the principles that underlie their inter-reduction.

3.2.1 Inner Workings of the Abstract Interpreter

[Figure 3.2](#) presents the internal architecture of [EVA](#). We comment on the diagram from bottom to top. Note that the three layers of

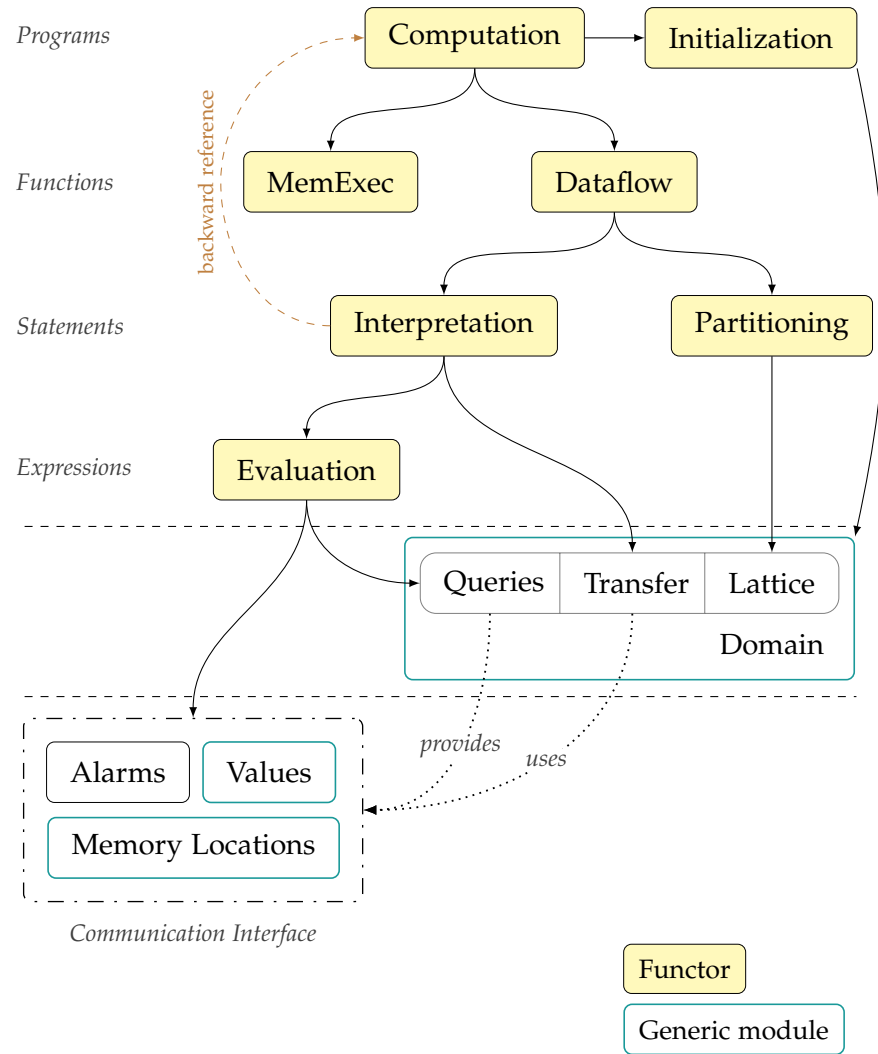


Figure 3.2: Architecture of the analyzer

Figure 3.1 are still shown, and that the figure focuses on the structure of the abstract interpreter. As described above, it relies on different abstractions to model the semantics of the C language.

The alarms are abstractions of the possible undesirable behaviors of a program. They are listed in a fixed module that defines the scope of the verification made by the abstract interpreter. On the other hand, the value, location and state abstractions are parameters of an analysis. The abstract interpreter is made of a hierarchy of functors built upon these abstractions. In the diagram, yellow boxes are generic OCaml functors, and the plain arrows point out the module arguments of each functor.

The *evaluator* depends on the value (and location) abstractions, and the queries of an abstract domain. It is able to evaluate any expression into a value abstraction, or any address into a location abstraction, based on the abstract semantics provided by both value and location modules. It queries the abstract domain for initial value abstractions to work on, especially for abstractions of the possible value of variables. The evaluator also gathers the alarms produced during the evaluation of an expression.

The *interpretation* functor depends on an evaluator and the abstract transformers of the domain. It implements the generic transfer functions for C statements. It evaluates all the expressions involved in a given statement, emits the alarms issued by the evaluations, and then calls the abstract transformers of the domain. These transformers may use all the abstractions computed by the evaluations. For an assignment, this includes a value abstraction for the right expression, and a location abstraction for the lvalue being assigned. These abstractions have been computed cooperatively, using information coming from any abstract domain. Thus, if the analyzer is instantiated on a product of abstract domains, information may flow from a domain to another.

The *partitioning* functor expands an abstract domain into its power-set extension. Such an extension refines a domain by inferring a finite disjunction of abstract states at each program point, instead of only one abstract state. Most domains of abstract interpretation are convex: they can only represent convex sets of concrete elements. The convexity of domains leads to more scalable analyses, but prevents the verification of non-convex properties. Disjunctions may significantly improve the precision of an analysis: they overcome convexity limitations, and also mitigate the need for relational abstractions. In order to limit the cost of processing multiple abstract states in parallel, the number of disjuncts is bounded by a parameter of the analysis. The *dataflow* functor implements a forward dataflow analysis propagating a disjunctive set of states over a function body. Abstract states are propagated separately until their number reaches the configured limit; then, new states are systematically joined together. This functor

relies on a powerset domain, and on the generic transfer functions for one abstract state. It computes a fixpoint on the CFG of a function for the abstract semantics defined by the transfer functions.

A backward reference links the interpretation functor to the uppermost one. Indeed, the interpreter has to handle all the C statements, including function calls. The analysis of EVA is context-sensitive: the function calls are all symbolically inlined. On a function call, a new dataflow analysis is instantiated over the body of the called function, starting at its entry point with the abstract state from the call site. When the inner dataflow reaches a fixpoint, the analysis of the calling function continues with the abstract state computed at the return statement of the called function. The *memexec* functor serves as a cache for the entire analysis of a function. It stores the abstract state inferred at the end of a function body, according to the entry abstract state of the dataflow analysis. It is used to avoid performing a new analysis of a function body when the entry state would lead to the same results as with a previous input [Yak15].

In addition to the dataflow and to *memexec*, the *computation* functor also relies on an *initialization* functor. This last component produces the initial abstract state of the analysis, according to the global variables of the program, and to the arguments of its *main* function.

3.2.2 Combination of Abstractions

The abstract interpreter of EVA is modular: it can be instantiated over any compatible abstractions. Moreover, the abstractions it relies on may be a product of several datatypes. Indeed, the shared evaluation through value abstractions has been designed to let information flow between different abstract domains. The value and location abstractions may also be extended to enable new forms of communication. To this end, we provide generic combinators for state, value and locations abstractions. These three combinators are OCaml functors, from two abstraction modules to a new one, whose elements are pairs of both incoming abstractions. They preserve all the requirements needed to guarantee the soundness of the analyzer. Successive applications of these functors allow the combination of any number of abstractions.

The value and the location combinators make a standard partial reduced product. An inter-reduction function can be provided, to narrow the approximations made by each component using the other one. Otherwise, the new abstract semantics is simply the pointwise application of the semantics of each component.

The state combinator takes two abstract domains operating on the same value (and location) abstractions, and produces a new domain still operating on the same value (and location) abstractions. A query to a combination of abstract states returns the meet of the value ab-

stractions extracted from the query to each state. The abstract transformers are carried out pointwise on each abstract state, but each one may use value (and location) abstractions computed with the results provided by the other abstract state. Thus, there is no explicit reduction function for the product of states. Instead, their inter-reduction is achieved through the value (and location) abstractions, cooperatively computed by the evaluation of the expression and lvalues involved in the program statements.

The state combiner is limited to abstract domains understanding the same value abstractions. In order to link abstract domains built over different value abstractions, we also provide a generic converter: a functor that lifts an abstract domain over an abstraction V to an equivalent domain over an abstraction W , according to injection and projection functions between V and W .

More precisely, let \mathbb{D} be an abstract domain working on a value abstraction V , and E be an abstract domain working on a value abstraction W . We can combine them by:

1. combining the two value abstractions into the product $V \times W$. Conversion functions between the initial abstractions and the product are straightforward —the injection completes an abstract value with the top abstraction of the other module:

$$\begin{array}{ll} inj_V : V \rightarrow V \times W & proj_V : V \times W \rightarrow V \\ v \mapsto v, \top_W & (v, w) \mapsto v \end{array}$$

2. lifting separately each domain D and E to equivalent new domains D' and E' working on the value abstraction $V \times W$. This operation uses the conversion functions between value abstractions.
3. combining the domains D' and E' into the product $D' \times E'$.

The communication mechanism of [EVA](#) uses the value abstractions as implicit interface between state abstractions. This naturally limits the communication between domains understanding different value abstractions. However, a reduced product between value abstractions may achieve a communication even between such domains. Indeed, a value component provided by the query of the first domain may reduce the value component used by the transformer of the second domain. The following example illustrates this.

Example 6. We consider two non-relational memory domains D and E storing information about the possible C values of integer variables. D abstracts the value of a variable as an interval, while E abstracts it with a congruence. D and E naturally use respectively intervals and congruences as value abstractions.

Assuming a condition $x > 3$, where D and E provide respectively the interval abstraction $[0..12]$ and the congruence $0[3]$ for x :

- the standard arithmetic semantics of intervals and congruences leads to the abstractions $[4..12]$ and $0[3]$ for the variable x after the condition;
- the congruence information allows reducing the interval $[4..12]$ into $[6..12]$. This is the natural inter-reduction between interval and congruence.
- the domain transformers use the computed value abstractions to reduce their current states. The interval domain binds x to the new interval $[6..12]$.

To a certain extent, the architecture of [EVA](#) transfers the need of a reduced product to the lower layer of value abstractions, rather than to the more complicated layer of state abstractions. Finally, note that both abstraction combinators may be specialized, to enable further (direct) communication between specific modules.

3.2.3 *Instantiating the Abstractions*

The abstractions used for an analysis are parameters of a run of the abstract interpreter, provided on the command-line, or set through the graphical interface. The different abstractions are thus instantiated and combined at runtime, as well as the set of functors that forms the abstract interpreter.

3.3 STRUCTURING A COMBINATION OF DATATYPES

In this section, we present a mechanism to enable interacting with the components of a generic combination of OCaml types. We describe the constant inner shape of the datatype combination through a [GADT](#) [[JGo8](#)], and automatically build injection and projection functions between compound values (from the combination) and leaf values (at the root of the combination). We use such a structuring for the different abstractions of [EVA](#).

3.3.1 *Context and Motivation*

[EVA](#) is a modular analyzer, and can be instantiated with any abstraction fulfilling the requirements described in the next sections. Several abstractions are already provided, as well as generic combinators. By construction, the abstractions used during an execution of the analyzer are unknown, and are often a product of several underlying abstractions. However, there are real advantages in being able to extract a particular component from within these abstractions.

- Some optimizations of [EVA](#) rely on a particular abstraction, both for precision and performance. For instance, the evaluator can

Listing 3.1: Combination of Datatypes

```

1 module type Abstraction = sig type t [...] end
  module A1 : Abstraction = sig [...] end (* leaf abstraction *)
  module A2 : Abstraction = sig [...] end (* leaf abstraction *)
  [...]
5
  (* Abstraction combiner *)
  module Combiner (Left: Abstraction) (Right: Abstraction)
    : Abstraction with type t = Left.t * Right.t = struct
    type t = Left.t * Right.t
10  [...]
  end

```

proceed by disjunction on some specific value abstractions. The partitioning strategy can also be chosen according to the representation of the main abstract domain.

- Although the design of [EVA](#) enables some communication between abstractions, further collaboration may require a direct access to each other. This is typically the case when two linked state domains want to exchange information without using a new kind of abstract values.
- Tracking a particular abstraction during an analysis may ease its debugging.

Listing 3.1 presents the general situation we tackle. A generic module interface is defined, including an abstract type `t`; in our case, this is the signature of the abstractions used by the analyzer. Several leaf modules implement this interface. One or several functors are able to create new modules satisfying the signature from existing ones. Such a combiner builds a new datatype upon the types of its arguments. The objective is to automatically equip any abstraction module with functions enabling a direct interaction with its components—the leaf modules that are part of its composition. For a leaf module, this includes only itself.

3.3.2 Interface of a Combination

Listing 3.2 presents the External signature we seek. The keys are polymorphic OCaml values identifying the type of the abstractions that may be part of the combination. Such a leaf abstraction, whose elements are of type `t`, must export a key carrying its type `t`, i.e. a key of type `t Key.k`. The `Key` module provides two functions: `create` and `eq_type`. The first one creates a new key from a string describing the abstraction to which the key will be linked. The second one compares

Listing 3.2: Keys and interface

```

1  type (_,_) eq = Eq : ('a,'a) eq | Neq

    module type Key = sig
        type 'a k
    5  val create : string -> 'a k
        val eq_type : 'a k -> 'b k -> ('a, 'b) eq
        val compare : 'a k -> 'b k -> int
    end

10 module type External = sig
    type t
    val get : 'a Key.k -> (t -> 'a) option
    val set : 'a Key.k -> 'a -> t -> t
end

```

two keys of respective types `a Key.k` and `b Key.k`. If they are the same, then `eq_type` returns an equality witness of the types `a` and `b`.

The `External` interface exports the type `t`, and two access functions `get` and `set`. If the type `t` is a combination of several datatypes, these functions allow interacting with its components, identified by their keys.

- `get` takes a key of type `a Key.k`, and:
 - if the type `t` is a combination of datatypes including the type `a` carried by the key, then `get` produces a new function that takes a value of type `t` and returns its component of type `a`.
 - otherwise, `get` returns `None`.
- `set` takes a key of type `a Key.k` and a matching value of type `a`, and inserts it in a value of type `t`, if possible —i.e. if the type `t` is a combination of datatypes including the type `a`. In this case, the previous value of type `a` in the combination is erased and replaced by the new value. Otherwise, the combination does not include a component of type `a`, and `set` is the identity function: it returns the combination as is, and ignores the value of type `a`.

3.3.3 Polymorphic Keys

If the leaf modules are all statically known, the keys can easily be implemented as a [GADT](#) (see [Listing 3.3](#)). However, this choice prevents creating keys for new abstractions dynamically. The architecture of [EVA](#) is intended to ease the introduction of new abstractions. We ultimately want to enable new abstractions to be implemented as other plugins of FRAMA-C, without modifying the analyzer itself.

Listing 3.3: Implementing keys with GADT

```

1 type 'a gadt_key =
  | A1 : A1.t gadt_key
  | A2 : A2.t gadt_key

```

Listing 3.4: Using extensible types to implement polymorphic keys

```

1 module Key : Key = struct
  type _ key = ..

  module type K = sig
    type t
    type _ key += Key : t key
  end

  type 'a k = (module K with type t = 'a)

  let create (type a) () =
    let module M = struct
      type t = a
      type _ key += Key : t key
    end in
    (module M : K with type t = a)

  let eq_type (type a) (type b) (x : a k) (y : b k) : (a, b) eq =
    let module A = (val x : K with type t = a) in
    let module B = (val y : K with type t = b) in
    match A.Key with
    | B.Key -> Eq
    | _ -> Neq
  end
end

```

This is why we use an extensible phantom type for keys. As we need to dynamically extend this type, we define each new constructor into a new module, allowing us to use the same constructor name each time. The keys are first class modules, each embedding a fresh constructor of the extensible type. Listing 3.4 exhibits their implementation. The idea comes from the module `Type_equal.Id` of the Core kernel library¹, which uses the same mechanism.

In practice, a key is a record gathering not only the first class module, but also an integer and a string. The string is the name of the key; it must be provided to the `create` function, and can then be printed for debugging purpose. A different integer is chosen for each key (by

¹ https://ocaml.janestreet.com/ocaml-core/latest/doc/core_kernel/Core_kernel/Type_equal.mod/

Listing 3.5: Exporting keys

```

1 module A1 : Abstraction = sig type t = [...] end
  let a1_key : A1.t = Key.create "A1"
  module A2 : Abstraction = sig type t = [...] end
  let a2_key : A2.t = Key.create "A2"

```

successive increment of a static counter). It serves as a hash of the key.

And now that we have polymorphic keys, a leaf abstraction module may export a key that identifies it, as in Listing 3.5.

3.3.4 Naïve Implementation

The polymorphic key allows a first straightforward implementation of the `External` signature, shown by Listing 3.6. For a leaf module, an access function compares the given key with the internal key of the domain, and behaves accordingly. The combiner requires that both argument modules also implement the `External` signature. Then, the access function of a combination simply calls that of the components.

Although simple, this implementation has several drawbacks:

- the code of the `get` and `set` functions is quite odd, and must be identically duplicated in each leaf module of abstractions;
- adding a new function based on the keys to the `External` signature requires modifying all existing domains.
- the functions are inefficient on a large combination of abstractions, as they systematically process all the underlying modules.

To address these issues, we only require the abstraction modules (leaf and combination) to provide a formal description of the internal structure of their datatype. More efficient implementation of the `get` and `set` functions for a particular module can then be deduced from its datatype structure.

3.3.5 GADT Structure of a Datatype

Listing 3.7 presents the structure type used to describe the internal shape of a datatype. A value of type `t structure` defines the structure of the datatype `t`. This definition can be directly the type `t` (the `Leaf` constructor, for a leaf module), or a pair of two datatypes described by structures (the `Node` constructor, for a combination of modules). The `Void` constructor is a valid structure for any type, but does not allow interacting with its components. It can be used for the components of a combination that do not need the functionalities

Listing 3.6: Naïve implementation of the External signature

```

1 module External_A1 : External with type t = A1.t = struct
  type t = A1.t
  let key : t key = Key.create ()

5  let get : type a. a key -> (t -> a) option = fun k ->
    match Key.eq_type key k with
    | Eq -> Some (fun t -> t)
    | Neq -> None

10  let set : type a. a key -> t -> a -> t = fun k t new_t ->
    match Key.eq_type key k with
    | Eq -> new_t
    | Neq -> t
  end

15 module External_Combiner (Left : External) (Right : External)
  : External = struct
    include Combiner (Left) (Right)

20  let get key =
    match Left.get key with
    | Some g -> Some (fun t -> g (fst t))
    | None -> match Right.get key with
      | Some g -> Some (fun t -> g (snd t))
      | None -> None

25  let set key (left, right) new_d =
    Left.set key left new_d, Right.set key right new_d
  end

```

Listing 3.7: Structure of a Datatype

```

1  type 'a structure =
    | Void : 'a structure
    | Leaf : 'a key -> 'a structure
    | Node : 'a structure * 'b structure -> ('a * 'b) structure
5
    module type Internal = sig
        include Abstraction
        val structure : t structure
    end
10
    module A1 = struct
        include A1
        let key : t key = Key.create "A1"
        let structure = Leaf key
15    end

    module A2 = struct
        include A2
        let structure = Void
20    end

    module Combiner (Left: Internal) (Right: Internal)
        : Internal = struct
        include Combiner (Left) (Right)
25    let structure = Node (Left.structure) (Right.structure)
    end

```

Listing 3.8: Dependent Maps

```

1 type map =
  | Empty : map
  | Node : map * 'a key * 'a data * map -> map

5 let rec find : type a. a Key.k -> map -> a Data.t option
  = fun key -> function
    | Empty -> None
    | Node (left, k, d, right, _) ->
      match Key.eq_type key k with
10   | Eq -> Some d
    | Neq ->
      find key (if compare key k <= 0 then left else right)

```

provided by the External signature. New constructors may also be added to describe other datatype layouts.

Then, any abstraction module has to fulfill the signature `Internal` of Listing 3.7, which simply adds to the signature of abstractions a value of type `t structure`, where `t` is the type of the abstractions. The structure of some leaf modules and of the combiner are also given in the listing.

For a leaf type `t` identified by a key `k` and a module `M` implementing the External interface, we call *accessors* for `t` in `M` the application of the `get` and `set` functions of a module to the key `k`. This builds the functions allowing the interaction with the component of type `t` from the module `M`, if it exists. The structure value of a module `M` allows the automatic generation of the relevant accessors for the datatypes that the module `M` indeed contains. These are the relevant accessors for the module, the others being dummy constant functions: for the types not included in a module, `get` always returns `None` and `set` is the identity function. Thanks to the structure, the relevant accessors can even be built once and for all, when the module is created (at compile-time for the leaf module, or at the application of the combiner functor). After their generation, we need to store the accessors and to make them available for use. As the type of an accessor depends on the type of the key, we use maps from keys to dependent data. Their implementation follows the one for usual maps, except that `find` needs the equality witness of keys (see Listing 3.8).

3.3.6 Automatic Generation of the Accessors

Finally, Listing 3.9 presents the generation of the relevant accessors of a module, based on the structure that it provides. The `Open` functor transforms a generic internal module, including a structure value that matches its internal type, into an external one. All relevant acces-

Listing 3.9: Accessors generation

```

1  module Open (M : Internal) = struct

    (* Getters *)
    type ('a, 'b) get = 'b key * ('a -> 'b)
    5  type 'a getter = Get : ('a, 'b) get -> 'a getter

    let lift_get f (Get (key, get)) = Get (key, fun t -> get (f t))
    let rec compute_getters :
        type a. a structure -> (a getter) KMap.t = function
    10  | Void -> KMap.empty
        | Leaf key -> KMap.singleton key (Get (key, fun (t : a) -> t))
        | Node (left, right) ->
            let l = compute_getters left and r = compute_getters right in
            let l = KMap.map (lift_get fst) l
    15  and r = KMap.map (lift_get snd) r in
            KMap.merge l r

    let getters = compute_getters M.structure
    let get (type a) (key: a key) : (M.t -> a) option =
    20  match KMap.find key getters with
        | None -> None
        | Some (Get (k, get)) -> match Key.eq_type key k with
            | None -> None
            | Some Eq -> Some get

    25  (* Setters *)
    type ('a, 'b) set = 'b key * ('b -> 'a -> 'a)
    type 'a setter = Set : ('a, 'b) set -> 'a setter

    30  let lift_set f (Set (key, set)) =
        Set (key, fun v b -> f (fun a -> set v a) b)
    let rec compute_setters:
        type a. a structure -> (a setter) KMap.t = function
    35  | Void -> KMap.empty
        | Leaf key -> KMap.singleton key (Set (key, fun v _t -> v))
        | Node (left, right) ->
            let l = compute_setters left and r = compute_setters right in
            let l = KMap.map (lift_set (fun set (l, r) -> set l, r)) l
            and r = KMap.map (lift_set (fun set (l, r) -> l, set r)) r in
    40  KMap.merge l r

    let setters = compute_setters M.structure
    let set (type a) (key: a key) : (a -> M.t -> M.t) =
        match KMap.find key setters with
    45  | None -> fun _ t -> t
        | Some (Set (k, set)) -> match Key.eq_type key k with
            | None -> fun _ t -> t
            | Some Eq -> set

```

sors are computed at the application of this functor, and stored in a dependent map. Then, we can pick up the accessor for a specific key; if the key is not in the map, the dummy accessor is returned instead. In both cases, the obtained accessor is efficient, as it knows statically the path to the requested leaf abstraction in the compound datatype. We detail the generation for the `get` function; the principles are the same for the `set` function.

The $(\text{'a}, \text{'b})$ `get` type denotes pairs of a key of type 'b and the `get` accessor for this key in the type 'a . The `compute_getters` function recursively gathers all the `get` pairs for a type 'a described by an `structure`, using a map from keys to these pairs (`KMap`). For a `void` structure, no accessor can be recovered. For a leaf structure, the only relevant accessor is the identity function for the key of the leaf. For a node structure, we compute recursively the accessors for both branches of the node; they are accessors for the sub-type of each branch, and we lift them on the new type denoted by the structure: pairs of elements from each branch. We then merge both accessor maps. `getters` is the map resulting from the application of `compute_getters` to the structure of the module argument of the functor. Then, the final `get` function amounts to search the corresponding accessor in the map. Note that we need again the type equality witness given by `Key.eq_type` for the correct typing of the function.

3.4 DEVELOPMENT AND CONTRIBUTIONS

[EVA](#) is not a completely new abstract interpreter: it is a major evolution of a pre-existing plugin of FRAMA-C called Value Analysis, and simply abbreviated as [VALUE](#). In this section, we briefly present this former abstract interpreter, emphasize the main innovations introduced with [EVA](#), and comment on its development.

3.4.1 *Evolution of the Abstract Interpreter*

Since its inception, FRAMA-C has contained an abstract interpreter, the [VALUE](#) plugin. This analyzer handles the subset of C99 commonly used in embedded software, and already gets precise results on such codes. It has also been considerably optimized for years to achieve scalability on large programs, and has indeed been successfully used to verify safety-critical industrial codes [[Cuo+12](#)]. [VALUE](#) features an intricate memory abstraction [[Kir+15](#), §4], able to represent efficiently and precisely both low-level concepts such as unions and bitfields, or high-level ones, such as arrays. This abstract domain is rich, but cannot infer relational properties. Interestingly, aggressive state partitioning is often sufficient to alleviate this limitation: the re-

lation information is instead carried out by the disjunction encoded by the multiple states.

More problematic was the fact that `VALUE` has been written around its abstract domain, resulting in a very tight coupling. The abstract domain was hard-wired in the analyzer, and adding new abstractions—relational or not—was not really possible. The primary objective of this work was to make the abstract interpreter adequately modular, in order to ultimately be able to complete the initial abstract domain with new abstractions, including (but not limited to) relational ones. Such a goal could only be achieved by a deep reorganization of the analyzer. This has required to specify the functions ascribed to the abstractions, and to design clear interfaces for them. Then we have rewritten the analyzer to comply with the new architecture, separating the existing code between a generic abstract interpreter and abstractions implementing the abstract semantics of `VALUE`. Thus, in the new `EVA` interpreter, the legacy abstract domain, inherited from `VALUE`, is only one of many abstractions.

However, a major challenge was also to preserve the already good performances of the `VALUE` analyzer, in terms of both precision and scalability. Yet, some optimizations and heuristics of the analyzer, which were crucial for its efficiency, relied on its internal representation of a program semantics, which is now the internal type of one abstract domain among others. These mechanisms could only be kept in `EVA` thanks to the architecture of abstractions described in Section 3.3. Indeed, this allows the generic abstract interpreter to extract from an abstract domain the component corresponding to the legacy `VALUE`, and to apply the algorithms accordingly. Such optimizations or heuristics, that need a specific abstraction to work, are simply disabled when the given abstraction is not available for an analysis. This may impact the results precision or analysis time, but not the soundness of the analysis. In particular, such optimizations and heuristics support the instance of trace partitioning and the cache of a function dataflow that was used by `VALUE`, and that have been ported for `EVA`.

In the end, `EVA` is a modular and extensible abstract interpreter. When instantiated with the legacy abstract domain inherited from `VALUE`, it provides the same functionalities, with equivalent performances, as `VALUE` did.

3.4.2 Contributions

The evolution of the abstract interpreter is not just about modularity; `EVA` also features a combination of abstractions and an implicit means of communication between them that achieve a new form of partial reduced products. Compared to the former abstract interpreter, the main novelties of `EVA` are:

- a modular architecture, enabling the instantiation of the analyzer on generic abstractions. It has also been designed to facilitate the introduction and the combination of new abstractions.
- a generic combination of abstractions that enables some direct interaction within the components.
- the direct cooperation of all abstractions to the emission of the alarms, that report the undesirable behaviors of a program. In [VALUE](#), alarms were directly emitted as a side effect of the application of any abstract semantics, which would have prevented an abstraction to reduce the alarms stemming from another one.
- the communication between state abstractions, through non relational abstractions of the possible C values of expressions — or of the possible locations of addresses.
- a shared evaluation of expressions in a product of abstract states, featuring an efficient backward propagation phase [[CC79b](#)], that depends on the produced alarms and on the reductions performed by the abstract domains. More systematic backward propagators have been written accordingly.
- the introduction of several new abstractions, including:
 - a binding for the numerical abstract domains provided by the APRON library;
 - an abstract domain of symbolic equalities;
 - bitwise value and state abstractions, a domain of symbolic locations and a gauge domain, all contributed by Boris Yakobowski.

From the diagram of Figure [3.2](#), the initialization, partitioning, data-flow and memexec functors are mostly inherited from the former [VALUE](#) interpreter. We have only modified them to turn them into generic functors, without modifying their algorithms. The other parts of the abstract interpreter are essentially new.

3.4.3 Development

The OCaml implementation of the work presented in this manuscript is available as the [EVA](#) plugin of the silicon version of FRAMA-C², released on December 2016. The source files can be found in the directory `src/plugins/value/`. The table of Appendix [C](#) presents the division in sub-directories, as well as the main files of the development, with a brief description of the features that each file implements, and references to the sections of this thesis that expound their role.

² Available at <http://frama-c.com/download-all-versions.html>

CONCLUSION

Within the [EVA](#) abstract interpreter, the internal hierarchy of abstractions follows the usual distinction between the expressions and the statements of imperative programming languages. Value abstractions approximate the semantics of expressions, while state abstractions—or abstract domains— approximate the semantics of statements. Multiple value abstractions are combined in a standard reduced product. Multiple abstract domains exchange information using the value abstractions as a shared communication interface. Furthermore, an OCaml [GADT](#) encodes the inner shape of the combination of abstractions, enabling direct interactions with its components.

The architecture and the features described in this chapter seem suitable for the development of the abstract interpreter of any imperative language. As a plugin of FRAMA-C, [EVA](#) is dedicated to the analysis of C programs. The next chapter introduces the C programming language, and formalizes the concrete semantics used to establish the soundness of our analysis.

The C programming language [KR78], created by Thompson and Ritchie around 1970, is nowadays one of the most used in the world. It remains the choice language for embedded and safety-critical programs. Yet, some of the distinctive features that explain its success also make it difficult to analyze. To cope with some of these impediments, FRAMA-C is built on an extended version of the C Intermediate Language (CIL) [Nec+02], an open-source front-end for the C programming language that facilitates program analysis and transformation. Basically, CIL is a simplified subset of C featuring a few core constructs with a clean semantics.

However, CIL is still huge and overly complicated for our needs. This thesis is not intended to present the exhaustive interpretation of the C semantics by EVA, but only some of its most interesting concepts. While EVA works directly on the CIL AST, we formalize its analysis on a simplified language named `clike`. We design `clike` as a toy language, but close enough to C for the features we want to focus on. On the one hand, `clike` enables the encoding of all the pointer arithmetic allowed in C. It is even more permissive than a strict interpretation of the C standard: its pointer values can be handled as standard integers. This allows the analysis of more real-world programs. On the other hand, it does not include the most complex memory structures, such as unions or bitfields. They are however supported by EVA, in particular by the main abstract domain inherited from the VALUE plugin.

The remainder of this chapter presents the C language through some important facets of its *spirit*, its underlying principles. It also outlines the simplifications offered by CIL. It then describes the syntax and the principles of our own language, and finally formalizes its concrete semantics, emphasizing the processing of the pointer arithmetic.

4.1 THE C LANGUAGE

This section provides an overview of the C language, as described in the standard, and of the CIL front-end. It especially focuses on some key features of the language that are crucial for programmers, and that a static analyzer cannot overlook.

4.1.1 *The C Standard*

The current official specification of the syntax and the semantics of the C language is the C11 standard [C11], issued by the ISO. This document is written in natural language and does not use a mathematical formalization to characterize the C semantics. Although intended to be well defined, this natural language text can only provide an informal semantics description, that may be ambiguous for the readers and sometimes leads to misinterpretation. This is all the more a serious drawback that the C standard is a contract between compiler implementors and program writers. Misunderstanding about the standard meaning often results in programming bugs, when the execution of a program does not behave as its author planned to.

Another difficulty of formally analyzing the C semantics lies in the large number of constructs included in the C language. For instance, it supports many control flow patterns, some of them being redundant: *for* iterations in addition to the *while* loops, *switch* disjunctions in addition to *if* conditionals, and several jump statements, either restricted (*break* and *continue*) or unrestricted (*goto*). It also provides a large choice of data types, including recursively defined types, enumerations, variable length arrays, unions, bitfields... Moreover, any expression may embed side-effects, which change the state of the execution environment when evaluating the expression. The unspecified evaluation order even leads to non-determinism. All this diversity of functionalities is a blessing for the programmers, but also a curse for analyzers.

To handle more easily the breadth of the C semantics, FRAMA-C relies on the CIL framework.

4.1.2 *C Intermediate Language*

CIL [Nec+02; Cil] is the Abstract Syntax Tree (AST) of FRAMA-C. It was originally a front-end for the C programming language aiming at facilitating program analysis and transformation. It consists in a highly-structured subset of C, provided with a cleaner semantics. It simplifies C programs by clarifying ambiguous constructs and removing redundant ones. It parses, type-checks and transforms any valid C program into a structured and typed AST that the FRAMA-C plugins—including EVA— can analyze. Among the most important simplifications performed by CIL are the following:

- All forms of loop constructs are transformed into an infinite loop *while(1)* with an explicit break statement for termination.
- Shortcut evaluations of boolean expressions and ternary conditional operators *?:* are also compiled into explicit conditionals.
- All functions bodies include a single return statement.

- Explicit casts are inserted for all necessary promotions and conversions of types.
- The side-effects of expressions are moved into separate statements. Non-determinism is handled by specific *unspecified sequence* of statements, whose order of execution is not specified.
- The prototype of never called functions are removed.

All these transformations help the analyzer, both by reducing the number of constructs to deal with, and by making explicit some intricate behaviors of the C standard (like side-effects). The [CIL](#) version of FRAMA-C also supports the logical annotation of a program through the ANSI/ISO C Specification Language ([ACSL](#)) [[ACSL](#)]. It is a behavioral formal language for C programs, inspired by JML [[Lea+08](#)]. [ACSL](#) can express a wide range of functional properties through assertions attached to a statement or function contracts, either partial or complete.

Despite the efforts made toward a clearer semantics, [CIL](#) remains a large piece of work. Although the [EVA](#) analyzer handles most part of [CIL](#), the scope of this thesis does not include the exhaustive interpretation of the C semantics. This is why this thesis is based on a smaller, simplified language. However, some key features of the C language are too important to be overlooked by our simplifications. We detail these features and their consequences before defining our language.

4.1.3 The C Spirit

The complexity of analyzing C programs is not only due to the lack of mathematical formalization of the semantics, or to the wide range of program constructs provided by the language. Some difficulties also arise directly from some of the goals pursued by the language, and thus cannot be avoided by simplifying it, or by considering only a subset of its semantics, without depreciating the relevance of the analysis.

The main principles that have guided the standard committee are underlined in the introduction of the C99 Rationale [[C99](#)]. In particular, the committee states that a major goal was to preserve the traditional *spirit of C*, which essence lies in the *underlying principles upon which the C language is based*. This spirit is summarized by a few sentences, including:

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Make it fast, even if it is not guaranteed to be portable.

The C standard mostly achieves these objectives by underspecifying the language, and by providing both a low and a high level view of the memory. These design choices probably explain a part of the popularity of the C language, but are a burden for an abstract interpreter that has to cope with them.

4.1.3.1 *Underspecified Behaviors*

In order not to restrict compilers implementation, the C standard under-specifies the semantics of the language. It distinguishes three kinds of underspecifications:

- an *implementation-defined* behavior depends on the compiler implementation. Their precise specification is thus left to the compiler, which must document these behaviors. An example of such behavior is the size of integers in memory, even if the standard imposes a lower bound.
- an *unspecified* behavior is a choice between several fixed behaviors on some program constructs, defined by the standard. A compiler does not need to document its choices, that may depend on the context. For instance, the arguments of a function can be evaluated in any order at each call site.
- an *undefined* behavior occurs on non-portable or illegal program constructs, and on use of erroneous data, for which the standard imposes no requirements. In case of undefined behaviors, the execution of a program can do literally anything, including termination, crash or unforeseeable results. Undefined behaviors include dereferencing a null pointer, signed integer overflows and divisions by zero.

The underspecification of the C standard gives a compiler more leeway to efficiently compile C programs. Implementation-defined behaviors have been used to ease the support of various hardware architectures, and nowadays C runs on almost any computer in the world. Unspecified behaviors let the compiler choose freely the most convenient way to process some constructs. Undefined behaviors spare the compiler to check for error cases, and thus allow the generation of more efficient code.

However, underspecification is also a burden for program writers. To be portable, a program must not depend on some specific behaviors defined by a compiler, or on the choice of unspecified behaviors made by the compiler. Moreover, a program is correct only if it is free of undefined behaviors. However, such behaviors can be hard to detect at runtime, as their actual effect during the execution is unpredictable. Sometimes, the implementation of undefined behaviors by a compiler does not raise any error, and behaves exactly as expected by the programmer. But a compiler may change its implementation

Listing 4.1: Detection of undefined behaviors

```

1 int main () {
    int a, b, c;
    b = 1;
    *((int *)&a + 1) = 0;
5   c = 1 / b;
    return c;
}

```

choices at any time without notice. A program involving such behaviors is not correct, even if its execution reveals nothing wrong.

Example 7. Let us consider the program of listing 4.1. At line 4, the addition between the pointer `(int*)&a` and the integer 1 is well defined: the pointer behaves the same as a pointer to the first element of an array of length one with `int` type, and the expression `(int*)&a+1` points one past the last element of this array [C11, Section 6.5.6]. However, the use of the result of an addition that points just beyond an array object as the operand of the `*` operator is an undefined behavior [C11, Annex J.2]. The standard allows the computation of such pointers that cannot be dereferenced because of their convenience when looping through arrays. Still, according to the C standard, the program of listing 4.1 exhibits an undefined behavior, and thus its execution behaves arbitrarily.

In practice, most compilers dereference the address resulting from the addition without further check. The location that is assigned at line 4 thus depends on the memory layout between the program variables, that can differ at each execution. If the variable `c` is adjacent to `a` in memory, then the variable `c` is written at line 4, which does not impact the remainder of the program. If the variable `b` is adjacent to `a` in memory, then the variable `b` has the value 0 at line 5, which may cause a crash when dividing by `b` (or a flight simulator to launch). If the memory location adjacent to `a` is invalid, then the program will probably fail from line 4.

Writing correct C programs requires an extensive knowledge of the underspecifications, which are pervasive in the language: the C11 version of the standard includes no less than 203 kinds of undefined behaviors [C11, Annex J.2]. In this regard, automatic analyzers able to ensure the absence of undefined behavior in a program can provide a valuable assistance.

These analyzers have to handle the three kinds of underspecifications used by the standard. The concrete semantics of EVA follows the solutions commonly used in abstract interpretation. The implementation-defined behaviors are mostly parameters of the analysis: a program is then verified with respect to a given target architec-

Listing 4.2: Object representation of pointers

```

1 int main () {
    int y = 1, x = 0;
    int *p = &x + 1, *q = &y;
    if (memcmp(&p, &q, sizeof(p)) == 0)
5     printf("%d %d %d\n", *p, *q, p == q);
6 }

```

ture and a specific compiler. The unspecified behaviors are modeled by non-determinism: all possible implementation choices are considered by the analysis. Finally, the undefined behaviors are considered as errors that abort program executions. The undefined behaviors that EVA currently detects are listed in Section 5.1.

4.1.3.2 Low-level and High-level View of the Memory

The desire of providing the programmers with all the tools they may need explains the duality of the C memory model. The C standard exposes both a low-level and a high-level view to the memory:

- a high-level view by means of the effective types of structured objects, including arrays, structs and unions [C11, Section 6.2.5];
- a low-level view by means of the untyped byte representations of values [C11, Section 6.2.6].

Any value of a type τ is represented as a sequence of bytes, and can be interpreted as an array of n unsigned single-byte characters, where n is the size in bytes of an object of type τ . This byte interpretation is called the *object representation* of the value. An object can be read and written either through typed expressions or through byte manipulations of their object representation. The definition of the object representations of values is partly imposed by the standard, and partly left to the compiler. A formal analysis of C programs requires to reconcile the low-level and the high-level world. Greater difficulties arise when these two worlds conflict, i.e. when a typed value and its object representation are inconsistent.

Some object representations do not match a typed value, and are called *trap* representations [C11, 6.2.6.1 §5]. They can be manipulated byte to byte, but their access through a typed lvalue is undefined. A value may have also several object representations, and thus the equality between two values x and y of type τ does not imply the equivalence of their byte interpretations, that can be checked by `memcmp(&x, &y, sizeof(τ))`. The standard states that the equality $x == y$ does not necessarily imply that x and y have the same value; other operations may distinguish between them [C11, 6.2.6.1 §8].

Even more unsettling, it appears that the equivalence of the object representation may not guarantee the equality between values. The program of listing 4.2 may exhibit such a behavior. If the variable y is adjacent to x in memory, then the two pointers p and q point to the same memory address $\&y$. Therefore, they may have the same object representation (unless the compiler stores additional information in the pointer representation, such as the origin of the pointer). However, as seen at example 7, the dereference of p is an undefined behavior and may behave arbitrarily, unlike the dereference of q . In practice, the program is compiled without warning by gcc 6.2.1 with the options `-O1 -Wall`, and it prints '1 1 0' for $*p$, $*q$ and $p==q$, meaning that the pointers have the same object representation, different values, and that they point to the same variable.

In practice, such examples are corner cases that can be forbidden without a significant loss of expressiveness: a program behavior should not depend on the variable layout in memory. However, both kinds of memory accesses offered by the C standard are important to programmers, and a realistic analyzer cannot omit the features that support the low-level world, such as the pointer arithmetic or the connection between pointers and arrays. Therefore, the concrete semantics that we use in EVA forbids any discrepancy between typed values and their objects representations, but provides:

- both consistent low-level and high-level views of the memory, linked by bijective interpretation functions that are parameters of the analysis. This is described in Section 4.2.2.
- pointer arithmetic where pointer values are integer addresses, and can be manipulated as such. However, the language prevents jumping from a variable to another, establishing a separation of the variables in memory. This is described in Section 4.3.

4.2 A C-LIKE LANGUAGE

Although EVA works on CIL and thus supports a large part of the C language, this thesis does not aim at detailing its interpretation of the whole C semantics. It rather intends to present and formalize the general concepts that underlie the abstract interpreter architecture. We believe that these concepts are worthwhile regardless of the analyzed language, but they also take into account the specificities of the C standard described by the above section. We thus base our work on a simplified language, close enough to CIL, provided with a clearer semantics, and that still complies with the principles of the C standard stated before. This language is inspired by that of Miné [Mino6b]. We name it clike.

4.2.1 Syntax

Figure 4.1 introduces the syntax of `clike`.

A program operates over a fixed, finite set of variables $x \in \mathcal{X}$ in memory. Each variable has a type, and can hold only a value of its type. The type of a variable x is given by a builtin function `typeof(x)`. Our language manipulates values and objects of different types:

- *arithmetic* types denote values in different subsets of \mathbb{Q} , including signed or unsigned integers types, real floating-point types, and the basic type `char`.
- a τ -*pointer* type describes a reference to an entity of type τ in the memory. A pointer value is either the `NULL` pointer or a pair $(\&x, i)$ of a variable x and an integer, byte-expressed, offset i such that $0 \leq i \leq \text{sizeof}(x)$.
- Arithmetic and pointer types are collectively called *scalar* types. We denote by $\overline{\mathbb{V}}_\tau$ the set of values ranged by a scalar type τ , and by $\overline{\mathbb{V}}$ the set of all scalar values.
- Finally, inhabitants of *aggregate* types can be constructed from objects of different types, through arrays of fixed size and structures:
 - $\tau[n]$ is the type of an array of n objects of type τ , adjacent in memory.
 - $\{field_1 : \tau_1; \dots; field_k : \tau_k\}$ is the type of a structure made of k adjacent boxes in memory, such that the n^{th} box has the name $field_n$ and contains an object of type τ_n .

Definition 19 (pointer value). A not null pointer value is a pair of a variable x and an integer i such that $0 \leq i \leq \text{sizeof}(x)$. It is written $(\&x, i)$.

The expressions of `clike` are typed. An *expression* of our language is either:

- a constant, namely a scalar value. Among them, the address of a variable x is $(\&x, 0)$, but we often write it $\&x$ for readability.
- the application of an n -ary operator \diamond on expressions. Operators in OP include arithmetic operations, equality and relational comparisons, left and right shifts, bitwise operators and casts into a scalar type. They follow the same syntax as in C but are specialized according to their type. For instance, the addition is derived into several operators $+_\tau$ carrying the type τ of scalar values on which it can be applied.
- the dereference $*_\tau a$ of an address a . It reads a value of a *scalar* type τ at the memory location denoted by a .

Types:

$$\begin{aligned}
 integer &::= \text{char} \mid \text{short} \mid \text{int} \mid \text{long} \mid \text{long long} \\
 float &::= \text{float} \mid \text{double} \mid \text{long double} \\
 arith &::= \text{char} \mid (\text{signed} \mid \text{unsigned}) \text{ integer} \mid \text{float} \\
 scalar &::= \text{arith} \mid \text{type ptr} \\
 \tau \in \text{type} &::= \text{scalar} \mid \tau[n] \mid \{field_i : \tau_i\}_{i \leq n} \quad n \in \mathbb{N}
 \end{aligned}$$

Variables: $x \in \mathcal{X}$

Field names: $field \in F$

Values:

$$\begin{aligned}
 \bar{\mathbb{V}}_{\text{int}} &\subset \mathbb{Z} \\
 \bar{\mathbb{V}}_{\text{float}} &\subset \mathbb{Q} \\
 \bar{\mathbb{V}}_{\text{ptr}} &\triangleq \{(\&x, i) \mid x \in \mathcal{X}, 0 \leq i \leq \text{sizeof}(x)\} \\
 &\quad \cup \{\text{NULL}\} \\
 \bar{\mathbb{V}} &\triangleq \bigcup_{\tau \in \text{scalar}} \bar{\mathbb{V}}_{\tau}
 \end{aligned}$$

Expressions and addresses:

$$\begin{aligned}
 e \in \text{expr} &::= cst & cst \in \bar{\mathbb{V}} & \text{scalar constant} \\
 &\mid \diamond(e^n) & \diamond \in \text{OP} & \text{operators} \\
 &\mid *_\tau a & & \text{dereference} \\
 a \in \text{addr} &::= e \mid a.field \mid a[e]
 \end{aligned}$$

Statements:

$$\begin{aligned}
 stmt &::= *_\tau a := e & \text{assignment} \\
 &\mid *_\tau a \leftarrow *_\tau a & \text{copy} \\
 &\mid e == 0? & \text{test} \\
 &\mid \text{enter_scope}(x) \\
 &\mid \text{exit_scope}(x)
 \end{aligned}$$

Figure 4.1: Syntax of the clike language

The dereference $*_{\tau}a$ of the address a is called a *lvalue*. An *address* is either:

- a direct expression of pointer type;
- an address a plus an offset *field* for a field in a structure: $a.\textit{field}$;
- an address a plus an integer expression i for the offset of a cell in an array: $a[i]$.

In `clike`, an access to the memory is always an explicit dereference of an address. This design provides a clear separation between the computation of a memory location from an address, and the reading of the memory at this location. This leads to a unified formal definition of the semantics of any memory access. In this semantics, reading the value of a variable x of type τ is achieved by the dereference of its address $*_{\tau}(\&x)$. Similarly, the access to the n^{th} cell of an array t is the dereference $*_{\tau}(\&t[n])$. It is important to remember that in the general case where e is an arbitrary expression of pointer type, $e[n]$ is an address, and the access to the cell is made by $*_{\tau}(e[n])$. For readability however, we write most of our examples using the C syntax: we simply write x for the dereference $*_{\tau}(\&x)$, and $t[n]$ for an access to an array cell.

As in `CIL`, the expressions have no side-effects. All modification of the environment or observable behaviors are carried out by statements. A *statement* is either:

- an assignment from the value of an expression to a lvalue;
- the direct copy of the content of a lvalue location into another;
- a test that halts the execution when a condition is not satisfied;
- the entry in scope or the exit from scope of a variable.

Following our mathematical representation of programs defined in Section 2.1, a program is a graph whose nodes are integer-numbered program points and whose edges are labeled by statements. A program has a single entry point, which by convention is the node 0, and a single ending point. Such a graph is a `CFG` and is represented by a set of triples (source node, statement, destination node). However, for clarity, we write code using a C-like syntax.

As explained in Section 2.1, the graph structure of `clike` clearly encodes the various control-flow constructs of the C language, as conditional branches, switch disjunctions, loops or goto statements. In this way, they are directly managed by the fixpoint engine of `FRAMA-C`. Figure 4.2 gives an example of C code, its conversion in a `clike` graph structure, and the exact translation of each statement in the `clike` syntax. The side-effect `++i` of the conditional expression at line 3 is converted into the assignment statement between nodes 2

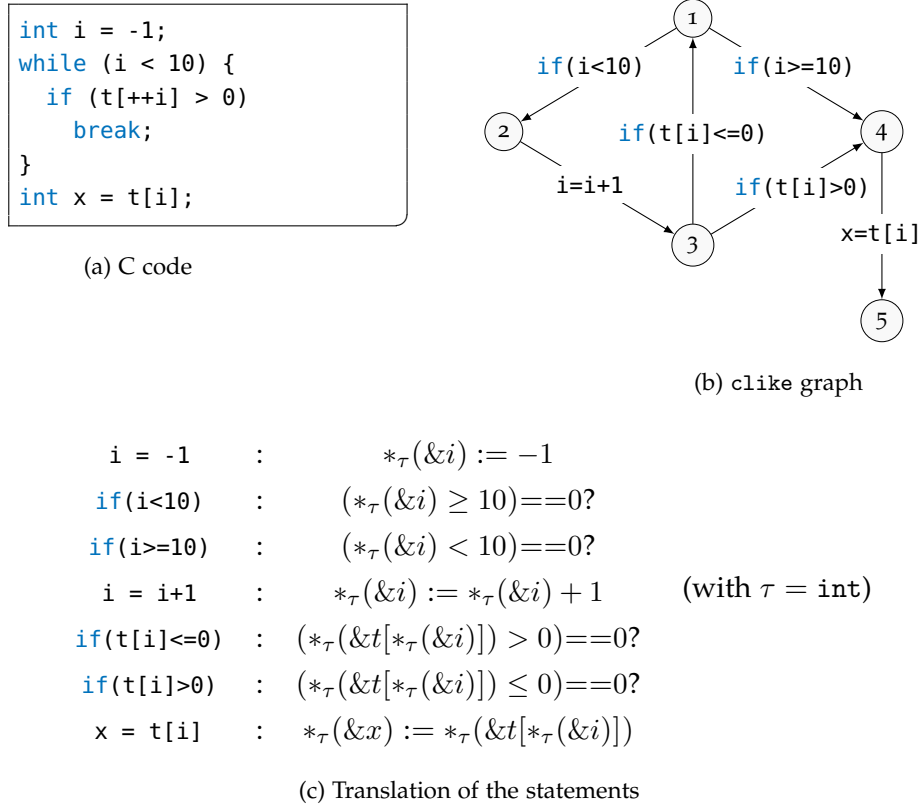


Figure 4.2: From C to clike

and 3, before the translation of the conditional statement. The *while* loop becomes simply the graph loop between nodes 1 (the loop entry point) to 3. The natural loop exit is the edge between nodes 1 and 4, and the break exit is the edge between nodes 3 and 4.

The remaining of this section gives meaning to the values and expressions of *clike*, but the concrete semantics that we will actually use is formally defined in the next section.

4.2.2 Representation of Values in Memory

The semantics of *clike* is designed to allow the dereference $*_{\tau}a$ of a value of any scalar type τ at any address a , provided that a is a *valid* address for this type (the validity of a pointer is defined later). In particular, the value of any variable can be read as an array of *char*, as in the C language. However, defining such a semantics firstly requires to formalize the connections between values of different types. This section defines the representation of the values in memory, and their interpretation as a scalar type.

	notation	signature
for an arithmetic type τ	ϕ_τ	$(\overline{\mathbb{V}}_{\text{bytes}})^{\text{sizeof}(\tau)} \rightarrow \overline{\mathbb{V}}_\tau$
for variable addresses	ϕ_{address}	$(\overline{\mathbb{V}}_{\text{bytes}})^{\text{sizeof}(\text{ptr})} \rightarrow \mathbb{N}$
memory layouts	θ	$\mathcal{X} \rightarrow \mathbb{N}$
for the pointer type ptr	ϕ_{ptr}	$(\overline{\mathbb{V}}_{\text{bytes}})^{\text{sizeof}(\text{ptr})} \rightarrow \overline{\mathbb{V}}_{\text{ptr}}$

$$\phi_{\text{ptr}}(\&x, i) = \phi_{\text{address}}(\theta(x) + i)$$

Figure 4.3: Interpretation functions and memory layouts

4.2.2.1 Interpretation of Values

The execution of a program is described by a sequence of concrete memories. A memory represents the content of each variable in \mathcal{X} as an array of single-byte characters in $\overline{\mathbb{V}}_{\text{bytes}} = \{0, 1, \dots, 255\}$. Concrete memories are kept untyped to allow casts between scalar types: an integer variable may store a pointer, and conversely. However, this requires some conversions between scalar values and sequences of bytes.

Indeed, the C values $\overline{\mathbb{V}}_\tau$ of a scalar type τ may be encoded on successive bytes, whose number and meaning depend on the hardware architecture, which we assume fixed. We assume given, for each scalar type τ :

- its size in bytes $\text{sizeof}(\tau)$. This function is naturally extended to variables and expressions, whose size is that of their type.
- a bijective interpretation function ϕ_τ defining the memory representation of its values. This function interprets a sequence of $\text{sizeof}(\tau)$ bytes as a value of type τ , and conversely for its inverse.

Definition 20 (Interpretation function). For a scalar type τ , an interpretation function is a bijective function ϕ_τ in:

$$\phi_\tau : (\overline{\mathbb{V}}_{\text{bytes}})^{\text{sizeof}(\tau)} \rightarrow \overline{\mathbb{V}}_\tau$$

The interpretation functions of arithmetic values in \mathbb{Q} are completely determined by the hardware architecture. However, the interpretation of pointer values depends on the memory layout. Each execution of a program induces a map θ from the variables to integers-numbered memory addresses. Such a map always fulfills some requirements.

Definition 21 (Memory layout). For a program P , a valid memory layout is an injective function $\theta : \mathcal{X} \rightarrow \mathbb{N}$ from variables to integers addresses such that:

- the integer memory address of a variable is strictly positive (the integer 0 is used for the representation of the NULL pointer):
 $\forall x \in \mathcal{X}, \theta(x) > 0$
- the contents of different variables do not overlap in memory :
 $\forall x, y \in \mathcal{X}, \theta(y) > \theta(x) \Rightarrow \theta(y) - \theta(x) > \text{sizeof}(x)$
 The comparison is strict to prevent two variables to be placed contiguously in memory. This is used later to always disambiguate pointers to $\&y$ from pointers to $(\&x, \text{sizeof}(x))$.
- the content of the variables fits in memory:
 $\forall x \in \mathcal{X}, \theta(x) + \text{sizeof}(x) < 2^{\text{sizeof}(\text{ptr})}$

Notation 1. For a program P , the set of valid memory layouts is written Θ_P .

Note that the requirements for the valid memory layouts limit the number and the size of the variables a program can use, according to the size of the pointer type.

The interpretation of pointers is then the combination of the memory layout θ and an interpretation ϕ_{address} of integer memory addresses. The NULL pointer is always interpreted as the integer 0. A pointer $(\&x, i)$ made of a variable x and an offset i is interpreted as the integer address $\theta(x) + i$. Figure 4.3 summarizes the type of each interpretation function and the link between the interpretation of pointers and the memory layouts.

Definition 22 (Pointer interpretation). The interpretation ϕ_{ptr} of pointer values is defined as:

$$\begin{aligned}\phi_{\text{ptr}}(\text{NULL}) &= 0 \\ \phi_{\text{ptr}}(\&x, i) &= \phi_{\text{address}}(\theta(x) + i)\end{aligned}$$

Note that the second requirement of Definition 21 ensures that two different pointer values have different integer addresses.

Lemma 2. *In any memory layout θ and for any two pointer values $(\&x, i)$ and $(\&y, j)$ in $\bar{\mathbb{V}}_{\text{ptr}}$, we have:*

$$\theta(x) + i = \theta(y) + j \Rightarrow x = y \wedge i = j$$

Proof. We suppose that the two pointer values $(\&x, i)$ and $(\&y, j)$ are different. As a memory layout is injective, $\theta(x) \neq \theta(y)$. We assume $\theta(y) > \theta(x)$ (the other case is symmetrical). According to the second condition of Definition 21: $\theta(y) > \theta(x) + \text{sizeof}(x)$. We also know from the definition 19 of $\bar{\mathbb{V}}_{\text{ptr}}$ that $0 \leq i \leq \text{sizeof}(x)$ and $0 \leq j \leq \text{sizeof}(y)$. Finally, we get:

$$\theta(y) + j > \theta(x) + \text{sizeof}(x) \geq \theta(x) + i$$

□

Thus, the function $f(\&x, i) = \theta(x) + i$ is injective. As we require the interpretation functions to be bijective, ϕ_{address} is also bijective. The interpretation ϕ_{ptr} is then injective, and its inverse can be defined on the image $\phi_{\text{ptr}}(\overline{\mathbb{V}}_{\text{ptr}})$. Outside this image, the inverse fails.

Definition 23 (Inverse of the pointer interpretation). The interpretation of `sizeof(ptr)` bytes as a pointer value is defined as:

$$g(n) = \begin{cases} (\&x, i) & \text{if } \theta(x) + i = n \\ \emptyset & \text{if } \forall (\&x, i) \in \overline{\mathbb{V}}_{\text{ptr}}, \theta(x) + i \neq n \end{cases}$$

$$\phi_{\text{ptr}}^{-1} = g \circ \phi_{\text{address}}^{-1}$$

4.2.2.2 Indeterminate Values

During the execution of a program, the value of a variable is not always well defined in memory. On the one hand, the program variables have a scope, i.e. a part of the program where they can be referred to. Out of its scope, a variable has no content in memory, and any reference to it results in an error. On the other hand, at its entry in scope, the content of a variable is unspecified, until an assignment sets a first value to the variable.

To report these two indeterminate states of a variable in the memories, the bytes are enriched with two special values: `uninit` for the uninitialized variables and `none` for the out-of-scope variables. We distinguish these values to properly report the related undefined behaviors when they occur.

4.2.3 Validity of Pointers and Locations, Memories

We define here the *validity* of a pointer and the notion of memory location as a consecutive sequence of byte addresses. We then define concrete memories, on which the semantics of `clike` is based.

4.2.3.1 Pointer and Memory Locations

Definition 24 (Validity of a pointer). A pointer value $(\&x, i)$ is *valid* when it points inside the content of x : $0 \leq i < \text{sizeof}(x)$.

Definition 25 (Memory location). A *memory location* is a pointer value $(\&x, i)$ together with a type τ . We write it $\text{loc}_\tau(\&x, i)$. It represents the consecutive addresses of the $\text{sizeof}(\tau)$ bytes in memory starting at the pointer value. Those are bytes whose contents in memory form a value of type τ .

$$\text{loc}_\tau(\&x, i) \triangleq \{(\&x, i + n) \mid 0 \leq n < \text{sizeof}(\tau)\}$$

We denote by τ -location a location of type τ .

Definition 26 (Validity of a memory location). A location of type τ is *valid* when all the pointer values it represents are valid. In other words, a τ -location is valid when its pointer value is valid and when the offset plus the size being read (i.e. the size of τ) is smaller than the size of the variable.

Notation 2 (Valid τ -pointers). A pointer value is a valid τ -pointer when it defines a valid τ -location. The set of valid τ -pointers is written \mathcal{L}_τ . By extension, $\mathcal{L}_{\text{bytes}}$ is the set of valid pointer values.

$$\begin{aligned}\mathcal{L}_\tau &\triangleq \{(\&x, i) \mid x \in \mathcal{X}, 0 \leq i \leq \text{sizeof}(x) - \text{sizeof}(\tau)\} \\ \mathcal{L}_{\text{bytes}} &\triangleq \{(\&x, i) \mid x \in \mathcal{X}, 0 \leq i < \text{sizeof}(x)\}\end{aligned}$$

4.2.3.2 Concrete Memories

Finally, memories bind valid byte locations to byte values, enriched with the special values for uninitialized and out-of-scope variables.

Definition 27 (Concrete memories). A concrete memory m is a map from valid pointer values to the set of values $\overline{\mathbb{V}}_{\text{bytes}} \cup \{\text{uninit}, \text{none}\}$.

Notation 3. For a program P , the set of all concrete memories is written \mathfrak{M} .

$$\mathfrak{M} : \mathcal{L}_{\text{bytes}} \rightarrow (\overline{\mathbb{V}}_{\text{bytes}} \cup \{\text{uninit}, \text{none}\})$$

The set of concrete memories depends on the program through its variables, that define the set of valid pointer values. For readability, we choose to omit this dependency in the notation.

4.2.4 Evaluation of Expressions in a Memory

Figure 4.4 details the evaluation $\llbracket e \rrbracket_m$ of an expression e of scalar type in a memory m . It produces either a value in $\overline{\mathbb{V}}$ or an error Ω , in case of an illegal operation.

Constants are already values in $\overline{\mathbb{V}}$, and so require no treatment. The application of an operator $\diamond e^n$ relies on the semantics $\llbracket \diamond \rrbracket$ of the operator. It computes the resulting value or fails, according to the values of the arguments, and does not involve the memory. For the most part, our operator semantics follow that of the corresponding C operator, as defined in the standard. The undesirable behaviors that may occur in the C semantics are here failures leading to the error value. This includes for instance divisions by zero, integer overflows or invalid pointer manipulations. An exhaustive list of the undesirable behaviors tracked by EVA is available in Section 5.1.

The next rules of Figure 4.4 describe the computation of addresses and their dereference. The evaluation of the address of an array cell $t[e]$ shifts the address of t by the value of e , using pointer arithmetic.

$$\begin{aligned}
\overline{[\Diamond]} &: \overline{V}^n \rightarrow \overline{V} \uplus \{\Omega\} \\
\llbracket v \rrbracket_m &\triangleq v \quad \forall v \in \overline{V} \\
\overline{[\Diamond(e_1, \dots, e_n)]}_m &\triangleq \overline{[\Diamond]}(\llbracket e_1 \rrbracket_m, \dots, \llbracket e_n \rrbracket_m) \\
\llbracket t[e] \rrbracket_m &\triangleq \begin{cases} (\&x, i + \text{sizeof}(t[0]) \cdot j) & \text{if } \begin{cases} \llbracket t \rrbracket_m = (\&x, i) \in \overline{V}_{\text{ptr}} \\ \llbracket e \rrbracket_m = j \in \mathbb{N} \end{cases} \\ \Omega & \text{otherwise} \end{cases} \\
\llbracket s.\text{field} \rrbracket_m &\triangleq \begin{cases} (\&x, i + \text{offsetof}(\text{field})) & \text{if } \llbracket s \rrbracket_m = (\&x, i) \in \overline{V}_{\text{ptr}} \\ \Omega & \text{otherwise} \end{cases} \\
\llbracket *_\tau a \rrbracket_m &\triangleq \begin{cases} \phi_\tau(\mathbf{m}(\&x, i), \dots, \mathbf{m}(\&x, i + \text{sizeof}(\tau) - 1)) & \text{if } \begin{cases} \llbracket a \rrbracket_m = (\&x, i) \in \mathcal{L}_\tau \\ \forall l \in \text{loc}_\tau(\&x, i), \mathbf{m}(l) \notin \{\text{uninit}, \text{none}\} \end{cases} \\ \Omega & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 4.4: Evaluation of expressions and addresses

Assuming that t evaluates to the pointer value $(\&x, i)$ and e evaluates to the integer j , the address of $t[e]$ is $(\&x, i + \text{sizeof}(t[0]) \cdot j)$. Computing the address of a structure field is similar, but relies on the `offsetof()` C macro to obtain the offset of the field in the structure type. This macro gives the offset (in bytes) of a given member within a structure type.

The dereference of an address fails if it is not valid for the read type τ : the address must evaluate to a valid τ -pointer $(\&x, i)$ in \mathcal{L}_τ . The dereference also fails if the memory contains some indeterminate value at any byte of the location $\text{loc}_\tau(\&x, i)$. Otherwise, the dereference reads the `sizeof`(τ) bytes starting at the address in memory, and interprets them as a value of type τ .

In order to simplify the writing of the dereference semantics, we introduce here a new notation about memories. We write $\mathbf{m}_\tau[(\&x, i)]$ the interpretation of the bytes of $\text{loc}_\tau(\&x, i)$ —i.e. the contents of these byte addresses in memory— as a value of type τ . If v is not a valid τ -pointer, then $\mathbf{m}_\tau[v]$ is the error value. If for a byte l of $\text{loc}_\tau(\&x, i)$, $\mathbf{m}(l)$ is uninitialized or out-of-scope, then $\mathbf{m}_\tau[(\&x, i)]$ is the error value. The value of the dereference $*_\tau a$ is then the τ -value stored in memory at the address a , namely $\mathbf{m}_\tau[\llbracket a \rrbracket_m]$.

Definition 28 (Load in memory and dereference).

$$\begin{aligned}
 \phi'_\tau(v_1, \dots, v_n) &\triangleq \begin{cases} \phi_\tau(v_1, \dots, v_n) & \text{if } \forall i, v_i \notin \{\text{uninit}, \text{none}\} \\ \Omega & \text{otherwise} \end{cases} \\
 m_\tau[v] &\triangleq \begin{cases} \phi'_\tau(m(\text{loc}_\tau(\&x, i))) & \text{if } v = (\&x, i) \in \mathcal{L}_\tau \\ \Omega & \text{otherwise} \end{cases} \\
 \llbracket *_\tau a \rrbracket_m &= m_\tau[\llbracket a \rrbracket_m]
 \end{aligned}$$

The next section uses these definition of values and expressions to develop a concrete semantics that remains consistent when pointer values can be handled as standard integers.

4.3 A CONCRETE SEMANTICS FOR CLIKE

In the clike language as in the C spirit, pointer values can be seen as integers, and can be converted into any scalar types through the interpretation functions. This tight connection between pointer values and integers makes the evaluation of expressions dependent of the address of variables—that is, dependent of the memory layout of program variables. This section illustrates this behavior, explains why it is undesirable, and formalizes a concrete semantics for clike that is independent of memory layouts. Following the guideline of Section 2.1, this concrete semantics is defined through the denotational characterization of expressions and statements.

4.3.1 Pointer Arithmetic and Memory Layout

Let us formally define the semantics of some operations about pointers, as they reveal interesting connections between our semantics of expressions and the memory layout of a program execution.

According to the C standard, arithmetic operations over pointers may only create *nearly valid* pointer values, namely pointers that point inside a variable or one byte past it. Reflecting this, a pointer value of clike is always nearly valid (definition 19). For instance, the addition of an integer to a τ pointer value adds to its offset the integer, multiplied by the size in bytes of the type τ of the pointed value. However, if the resulting pointer value is not nearly valid, the operation is undefined, and degenerates into the error value in our semantics¹. As we specifically banned overlapping between (even nearly) valid pointers values, this behavior prevents pointer arithmetic to jump from a variable to another. This is less general than in the C semantics, that state that two pointers compare equal if “one is a pointer to one past the

¹ Currently, EVA allows the computation of any invalid pointer, and only emits alarms when such pointers are dereferenced. We intend to change this behavior.

end of an object and the other is a pointer to the start of another object that immediately follows the first object in the address space” [C11, 6.5.9 §6].

$$\overline{[+_{\tau \text{ ptr}}]}((\&x, i), n) \triangleq \begin{cases} (\&x, i + \text{sizeof}(\tau) \cdot n) & \text{if } i + \text{sizeof}(\tau) \cdot n \leq \text{sizeof}(x) \\ \Omega & \text{otherwise} \end{cases}$$

However, the conversions between pointer and integer values are allowed, as long as they are consistent with the memory layout θ of the program execution. This is slightly more permissive than the C standard, in which such conversions are implementation defined, except for the NULL pointer that is equivalent to the integer constant 0. However, as stated in a footnote, “the mapping functions [for conversions between pointers and integers] are intended to be consistent with the addressing structure of the execution environment” [C11, 6.3.2.3 §5].

Definition 29 (Conversions between integer and pointer types).

$$\forall n \in \overline{\mathbb{V}}_{\text{int}}, \forall (\&x, i) \in \overline{\mathbb{V}}_{\text{ptr}}, \forall \tau \in \text{integer},$$

$$\begin{aligned} \overline{[(\tau)]}(\&x, i) &\triangleq \begin{cases} \theta(x) + i & \text{if } \theta(x) + i \in \overline{\mathbb{V}}_{\tau} \\ \Omega & \text{otherwise} \end{cases} \\ \overline{[(\tau \text{ ptr})]}(n) &\triangleq \begin{cases} \text{NULL} & \text{if } n = 0 \\ (\&x, i) & \text{if } n = \theta(x) + i \text{ and } (\&x, i) \in \overline{\mathbb{V}}_{\text{ptr}} \\ \Omega & \text{otherwise} \end{cases} \end{aligned}$$

Note that Lemma 2 ensures that the conversion from an integer to a pointer is well defined.

Moreover, integer arithmetic does not have the same constraints as pointer arithmetic. Thus, integer arithmetic on converted pointer values may lead to jumps between variables, according to a memory layout θ . The program on Figure 4.5 illustrates this possibility, on an architecture where unsigned integers can encode all pointer values. In a memory layout θ such that $\theta(b) = \theta(a) + 5$, this program is “correct” and assigns the value 42 to the variable b . In all other memory layouts, the value of $addr$ cannot be cast into a (nearly) valid pointer value at line 5, and the program is incorrect.

Thus, through the conversions between pointer values and integers, the concrete evaluation of expressions leads to program semantics that depend on the memory layout of a particular environment.

4.3.2 Concrete States Independent of the Memory Layout

An important asset of the C language is to feature both high-level and low-level access to the memory. While typed variables, structures and

```

1 void main() {
    int a, b;
    unsigned int addr = (unsigned int) &a;
    addr = addr + 5;
5  int *ptr = (int *) addr;
    *ptr = 42;
}

```

Figure 4.5: Jump between variables via integer arithmetic

arrays provide a high-level view of the memory, the objects stored in that memory can also be seen as an untyped contiguous sequence of bytes. Any values can then be read or written byte to byte. This low-level management can even be used for addresses and pointer values. As an example, on a standard architecture, the union structure of Figure 4.6 allows easy accesses to the low and high bytes of a pointer. Although programs seeking for portability should not include such constructs, they are perfectly legitimate in other contexts. This duality between low-level and high-level view of the memory needs to be reflected in our concrete semantics. Hence its dependencies to the interpretation functions, which link typed values to their byte representations.

However, a programmer should never assume anything about the memory layout of a program execution. A program based on an expected memory layout (as the one of Figure 4.5) must not be considered correct. Thus, the concrete semantics of our language should not depend on the addressing structure given by the θ function. Yet concrete memories have to depend on it: pointer values cannot be fully abstracted as long as they may be cut into bytes or converted into an integer.

The code snippet of Figure 4.7 illustrates this dependency through a quite strange but nevertheless legitimate way for writing a value in a variable x . The unsigned integer *half_addr* can not be represented as an abstract pointer value $(\&x, i)$. First, its integer offset would itself depend on the integer address of x . Moreover, the multiplication at line 4 has no meaning for a pointer value. However, the integers *half_addr* and *mod2_addr* are still related to the address of x , as the pointer x_ptr , computed from them, is a valid pointer to x .

Since memories are perforce related to the variable layout, our concrete semantics involve memories attached with the corresponding memory layouts. We remind the reader that for a program P , we denote by Θ_P the set of possible memory layouts for its execution; they are the maps from the program variables to integer addresses satisfying the conditions stated in Section 4.2.2. Then, a concrete state of our semantics is a function from memory layouts to concrete memo-

```

1 union ptr {
    struct ptr_s {
        low: short;
        high: short;
5    };
    void * ptr_p;
}

```

Figure 4.6: Pointer union

```

1 int x;
  unsigned half_addr = (unsigned)&x / 2;
  unsigned mod2_addr = (unsigned)&x % 2;
  int *x_ptr = (int *) (half_addr * 2 + mod2_addr);
5 *x_ptr = 42;

```

Figure 4.7: Pointer arithmetic through integers

ries. Such functions give the memory that occurs at a program point, according to the addressing structure of the program variables.

Definition 30 (Concrete State). For a program P , a concrete state S is a function from the memory layouts in Θ_P to concrete memories in \mathfrak{M} . The set of concrete states is:

$$\mathcal{S} : \Theta_P \rightarrow \mathfrak{M}$$

Thus, the value of some variables at a program point may also depend on the addressing structure. However, the *behavior* of the program should not. For instance, in the code of Figure 4.7, the integer variables *half_addr* and *mod2_addr* depend on the address of the variable x , but the final assignment at line 5 does not. The concrete state after this assignment would then be:

$$S = \lambda\theta. \left\{ \begin{array}{ll} x & \mapsto 42 \\ \text{half_addr} & \mapsto \theta(x)/2 \\ \text{mod2_addr} & \mapsto \theta(x) \bmod 2 \\ x_ptr & \mapsto \&x \end{array} \right\}$$

In contrast, in the code of Figure 4.5, the location of the final assignment at line 6 depends on the memory placement of the variables a and b . This later behavior should be strictly forbidden by the concrete semantics. We now formalize a concrete semantics of the language that meets those demands.

4.3.3 Concrete Semantics of Expressions

The concrete states of our semantics are functions from memory layouts to memories. As the value of expressions depends on the ad-

$$\begin{aligned}
\llbracket e \rrbracket^\Theta &: \mathcal{S} \rightarrow (\mathcal{V} \uplus \{\Omega\}) \\
\llbracket v \rrbracket^\Theta(S) &\triangleq \lambda\theta. v & \forall v \in \bar{\mathbb{V}} \\
\llbracket \Diamond(\vec{e}_i) \rrbracket^\Theta(S) &\triangleq \begin{cases} \lambda\theta. \llbracket \Diamond(\vec{e}_i) \rrbracket_{S(\theta)} & \text{if } \forall \theta \in \Theta_P, \llbracket \Diamond(\vec{e}_i) \rrbracket_{S(\theta)} \neq \Omega \\ \Omega & \text{otherwise} \end{cases} \\
\llbracket t[e] \rrbracket^\Theta(S) &\triangleq \begin{cases} \lambda\theta. \llbracket t[e] \rrbracket_{S(\theta)} & \text{if } \forall \theta \in \Theta_P, \llbracket t[e] \rrbracket_{S(\theta)} \neq \Omega \\ \Omega & \text{otherwise} \end{cases} \\
\llbracket s.field \rrbracket^\Theta(S) &\triangleq \begin{cases} \lambda\theta. \llbracket s.field \rrbracket_{S(\theta)} & \text{if } \forall \theta \in \Theta_P, \llbracket s.field \rrbracket_{S(\theta)} \neq \Omega \\ \Omega & \text{otherwise} \end{cases} \\
\llbracket *_\tau a \rrbracket^\Theta(S) &\triangleq \begin{cases} \lambda\theta. (S(\theta))_\tau[l] & \text{if } \forall \theta \in \Theta_P, \begin{cases} \llbracket a \rrbracket_{S(\theta)} = l \in \mathcal{L}_\tau \\ S(\theta)_\tau[l] \neq \Omega \end{cases} \\ \Omega & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 4.8: Concrete semantics of expressions and addresses

addressing structure, the concrete values are also indexed by the memory layouts. For a program P , they are functions from its correct memory layouts Θ_P to C values in $\bar{\mathbb{V}}$.

Definition 31 (Concrete value). For a program P , a concrete value V is a function from the memory layouts in Θ_P to C values in $\bar{\mathbb{V}}$. The set of concrete values is:

$$\mathcal{V} : \Theta_P \rightarrow \bar{\mathbb{V}}$$

The semantics of an expression in a concrete state S is its evaluation in each memory bound to a possible layout in S . If *any* of these evaluations fails, the semantics leads to the error value Ω . Otherwise, the semantics of an expression computes a value in $\bar{\mathbb{V}}$ for each memory layout—and the computed value may be different in each layout. However, some expressions not only affect the computation of values, but also impact the memory *behavior* of the program. This is the case for the addresses being dereferenced (for a read or a write operation), and for the conditions of if statements. We want such expressions to have the same value in a concrete state, regardless of the memory layout. We thus chose to enforce this rule in the concrete semantics of clike.

Figure 4.8 formalizes the concrete semantics of expressions and addresses on concrete values. It operates on concrete values in \mathcal{V} . It is denoted by $\llbracket \cdot \rrbracket^\Theta$, and derived from the expression evaluation in one

memory $\overline{\llbracket \cdot \rrbracket}$. Concrete states S are functions of type $\lambda\theta. m_\theta$, such as $S(\theta)$ is the memory m_θ in the layout θ . Concrete values V are functions of type $\lambda\theta. v$ such as $V(\theta)$ is the C value v in the layout θ . The semantics of an expression e take a concrete state and produce either a concrete value or the error case Ω .

$$\overline{\llbracket e \rrbracket}^\theta : \mathcal{S} \rightarrow (\mathcal{V} \uplus \{\Omega\})$$

The concrete semantics of a constant v is the constant function $\lambda_.v$. The concrete semantics of an operator application $\diamond(\vec{e}_i)$ is the pointwise application of the evaluation $\overline{\llbracket \diamond(\vec{e}_i) \rrbracket}_{S(\theta)}$ for all layouts θ , provided that none of them fails. The concrete semantics of addresses are defined likewise: they also fail if the evaluation fails in one of the memory layouts. A dereference also requires the address to be independent of the memory layout: for a dereference $*_\tau a$, the address a must evaluate to the same valid τ -location l in all the possible memory layouts of the program. All the memories of the state must also contain no indeterminate byte value at this location, allowing the dereference of the location to succeed in all layouts. Then, in the concrete state S , for each layout θ , the value of the dereference is the τ -value stored in the memory $S(\theta)$ at this location l . This value may depend on the memory layout.

The semantics of operators can also be lifted from C values in $\overline{\mathcal{V}}$ to functions in \mathcal{V} . Their concrete semantics become their pointwise application on the C value for each memory layout. A failure in one layout causes the error to be the result of the operation in all layouts.

Definition 32 (Concrete semantics of operators). The concrete semantics of operators $\overline{\llbracket \diamond \rrbracket}^\theta$ is defined from their previous semantics $\overline{\llbracket \diamond \rrbracket}$ as follows:

$$\begin{aligned} \overline{\llbracket \diamond \rrbracket}^\theta & : (\mathcal{V})^n \rightarrow (\mathcal{V} \uplus \{\Omega\}) \\ \overline{\llbracket \diamond \rrbracket}^\theta(\vec{V}) & \triangleq \begin{cases} \Omega & \text{if } \exists \theta \in \Theta_P, \overline{\llbracket \diamond \rrbracket}(\vec{V}(\theta)) = \Omega \\ \lambda\theta. \overline{\llbracket \diamond \rrbracket}(\vec{V}(\theta)) & \text{otherwise} \end{cases} \end{aligned}$$

The concrete semantics of operator application $\diamond(\vec{e}_i)$ can be equivalently redefined as the application of the concrete semantics of the operator \diamond on the concrete values V_1 to V_n stemming from the semantics of the expressions e_1 to e_n , provided that none of them fails.

Lemma 3.

$$\overline{\llbracket \diamond(e_1, \dots, e_n) \rrbracket}^\theta(S) = \begin{cases} \overline{\llbracket \diamond \rrbracket}^\theta(V_1, \dots, V_n) & \text{if } \begin{cases} \overline{\llbracket e_1 \rrbracket}^\theta(S) = V_1 \in \mathcal{V} \\ \dots \\ \overline{\llbracket e_n \rrbracket}^\theta(S) = V_n \in \mathcal{V} \end{cases} \\ \Omega & \text{otherwise} \end{cases}$$

Proof. Immediate from the above definition. \square

$$\text{loc}(\&x) \triangleq \text{loc}_{\text{typeof}(x)}(\&x, 0)$$

$$\frac{\forall \theta \in \Theta_P, \llbracket a \rrbracket_{S(\theta)} = l \in \mathcal{L}_\tau \quad \llbracket e \rrbracket^\Theta(S) = V \in \mathcal{V}}{S \vdash *_\tau a := e \Downarrow \lambda \theta. S(\theta) [\text{loc}_\tau(l) \mapsto \phi_\tau^{-1}(V(\theta))]}$$

$$\frac{\forall \theta \in \Theta_P, \llbracket a \rrbracket_{S(\theta)} = l \in \mathcal{L}_\tau \quad \forall \theta \in \Theta_P, \llbracket a' \rrbracket_{S(\theta)} = l' \in \mathcal{L}_\tau}{S \vdash *_\tau a \leftarrow *_\tau a' \Downarrow \lambda \theta. S(\theta) [\text{loc}_\tau(l) \mapsto S(\theta)(\text{loc}_\tau(l'))]}$$

$$\frac{\forall \theta \in \Theta_P, \llbracket e \rrbracket_{S(\theta)} = 0}{S \vdash e == 0? \Downarrow S} \quad \frac{\forall \theta \in \Theta_P, \llbracket e \rrbracket_{S(\theta)} \notin \{0, \Omega\}}{S \vdash e == 0? \Downarrow \emptyset}$$

$$\overline{S \vdash \text{enter_scope}(x) \Downarrow \lambda \theta. S(\theta) [\text{loc}(\&x) \mapsto \text{uninit}^{\text{sizeof}(x)}]}$$

$$\overline{S \vdash \text{exit_scope}(x) \Downarrow \lambda \theta. S(\theta) [\text{loc}(\&x) \mapsto \text{none}^{\text{sizeof}(x)}]}$$

(a) Operational semantics

$$\llbracket *_\tau a := e \rrbracket(S) \triangleq \begin{cases} \lambda \theta. S(\theta) [\text{loc}_\tau(l) \mapsto \phi_\tau^{-1}(V(\theta))] & \text{if } \begin{cases} \forall \theta \in \Theta_P, \llbracket a \rrbracket_{S(\theta)} = l \in \mathcal{L}_\tau \\ \llbracket e \rrbracket^\Theta(S) = V \in \mathcal{V} \end{cases} \\ \Omega & \text{otherwise} \end{cases}$$

$$\llbracket *_\tau a \leftarrow *_\tau a' \rrbracket(S) \triangleq \begin{cases} \lambda \theta. S(\theta) [\text{loc}_\tau(l) \mapsto S(\theta)(\text{loc}_\tau(l'))] & \text{if } \begin{cases} \forall \theta \in \Theta_P, \llbracket a \rrbracket_{S(\theta)} = l \in \mathcal{L}_\tau \\ \forall \theta \in \Theta_P, \llbracket a' \rrbracket_{S(\theta)} = l' \in \mathcal{L}_\tau \end{cases} \\ \Omega & \text{otherwise} \end{cases}$$

$$\llbracket e == 0? \rrbracket(S) \triangleq \begin{cases} S & \text{if } \forall \theta \in \Theta_P, \llbracket e \rrbracket_{S(\theta)} = 0 \\ \emptyset & \text{if } \forall \theta \in \Theta_P, \llbracket e \rrbracket_{S(\theta)} \notin \{0, \Omega\} \\ \Omega & \text{otherwise} \end{cases}$$

$$\llbracket \text{enter_scope}(x) \rrbracket(S) \triangleq \lambda \theta. S(\theta) [\text{loc}(\&x) \mapsto \text{uninit}^{\text{sizeof}(x)}]$$

$$\llbracket \text{exit_scope}(x) \rrbracket(S) \triangleq \lambda \theta. S(\theta) [\text{loc}(\&x) \mapsto \text{none}^{\text{sizeof}(x)}]$$

(b) Denotational semantics

Figure 4.9: Concrete semantics of statements

4.3.4 Concrete Semantics of Statements

Figure 4.9 presents the concrete semantics of statements, which is much more standard than the semantics of expressions. It operates on $\mathcal{S} \cup \{\Omega, \emptyset\}$, where:

- \mathcal{S} is the set of concrete states;
- Ω is a special state denoting an error —when the execution of a statement fails and the program aborts;
- \emptyset represents a blocking state: an assumption statement blocks the execution if the condition does not hold.

Figure 4.9 presents the equivalent operational and denotational semantics of statements. The operational semantics defines rules $S \vdash i \Downarrow S'$ that link an initial state S and a statement i to the result of the execution of i in S . The rules leading to the error state Ω are omitted in the figure. The denotational semantics of a statement i is a transfer function $\llbracket i \rrbracket$:

$$\llbracket i \rrbracket : \mathcal{S} \rightarrow \mathcal{S} \uplus \{\Omega, \emptyset\}$$

Again, a concrete state S is a function of type $\lambda\theta. m_\theta$, such as $S(\theta)$ is the memory m_θ in the layout θ . A sequence of concrete states throughout the statements describes an execution of the program for each possible memory layout.

ASSIGNMENT For an assignment $*_\tau a := e$ in the concrete state S :

- the address a must evaluate to the same valid τ -location $l \in \mathcal{L}_\tau$ in the memory $S(\theta)$, for all memory layout θ ;
- the evaluation of the expression e must produce a concrete value V such that in any memory layout θ , $V(\theta)$ is a valid value in \mathbb{V} .

Then, the concrete state after the assignment is the function S' such as for each layout θ , the memory $S'(\theta)$ is the previous memory $S(\theta)$ where the $\text{sizeof}(\tau)$ bytes of the location l are bound to the interpretation in bytes of the value $V(\theta)$. The $\text{sizeof}(\tau)$ bytes of the location l are denoted by $\text{loc}_\tau(l)$, and the interpretation in bytes of a value v of type τ is $\phi_\tau^{-1}(v)$. In each memory layout θ , the resulting memory is thus $S(\theta)[\text{loc}_\tau(l) \mapsto \phi_\tau^{-1}(V(\theta))]$.

Otherwise, the assignment fails.

COPY For a copy of lvalues $*_\tau a \leftarrow *_\tau a'$ in the concrete state S , both addresses —the one being copied and the one being assigned— should be independent of the memory layout. They must evaluate respectively to the same valid τ -locations l and l' in the memory $S(\theta)$, for all memory layout θ . Then, the concrete state after the assignment

is the function S' such as for each layout θ , the memory $S'(\theta)$ is the previous memory $S(\theta)$ where the $\text{sizeof}(\tau)$ bytes of the location l are bound to the content of the $\text{sizeof}(\tau)$ bytes of the location l' in $S(\theta)$. Thus, in each memory layout θ , the resulting memory is $S(\theta)[\text{loc}_\tau(l) \mapsto S(\theta)(\text{loc}_\tau(l'))]$. Otherwise, the copy of lvalues fails.

ASSUMPTION The truth value of the condition of a test should not depend on the memory layout. For a test $e==0?$ in a concrete state S :

- if for all layout θ , the condition e evaluates without error to the zero value in the memory $S(\theta)$, then the state S passes through the test unchanged;
- if for all layout θ , the condition e evaluates without error to a non-zero value in the memory $S(\theta)$, then the state S does not pass through the test, and the execution blocks—but does not fail;
- otherwise, the assumption fails, and the execution aborts. This happens when the evaluation of e fails for some layouts, or when it yields a zero value for some layouts, and a non-zero value for some others.

ENTRY IN SCOPE AND EXIT FROM SCOPE These two statements never fail. They change all the content of the involved variable in all the memories carried by the state. We write $\text{loc}(\&x)$ for the set of the valid byte locations of the variable x , which are $\text{loc}_{\text{typeof}(x)}(\&x, 0)$. All those bytes becomes `uninit` at the entry in scope of x , and `none` at its exit from scope. From a previous state S , the new state is then $\lambda\theta. S(\theta)[\text{loc}(\&x) \mapsto v^{\text{sizeof}(x)}]$, where $v \in \{\text{uninit}, \text{none}\}$.

CONCLUSION

Following the principles given in Section 2.1, the denotational semantics of the expressions and statements of `clike` defines naturally the collecting semantics of its programs. As in the common vision of the C language, this semantics allows addresses and pointer values to be handled as standard integers. However, the semantics also prohibits the behavior of a program to depend on the specific integer value of a variable address. This collecting semantics can then be soundly approximated in accordance with the abstract interpretation theory. The remaining of this thesis describes how this semantics can be practically handled by the hierarchy of abstractions implemented in `EVA`.

Part [iii](#) is dedicated to the abstract semantics of expressions:

- Chapter [5](#) formalizes the abstractions used for this purpose: the alarms, that report the illegal operations in expressions, and the

value abstractions, that approximate the result of the evaluation of an expression in a concrete state.

- Chapter 6 presents some efficient strategies to compute precise abstractions for complete expressions from partial information.

Part iv is dedicated to the abstract semantics of statements:

- Chapter 7 formalizes the state abstractions, on which are modeled the action expressed by a statement. It illustrates how the cooperative computation of alarms and value abstractions can assist a domain to precisely interpret the effect of statements.
- Chapter 8 describes the abstract domains that have been implemented within EVA, and the results of some experiments conducted to validate the design of the analyzer.

Finally, Part v is dedicated to the abstract semantics of traces. Chapter 9 presents a generic framework to improve the precision of standard abstract domains when two control-flow paths meet. At these points, domains are extended with conditional predicates that retain information coming from each path.

Part III

ABSTRACT SEMANTICS OF EXPRESSIONS

The abstract domains of [EVA](#) interact by exchanging properties about the expressions and the addresses involved in a program statement. These properties are expressed through specific abstractions of values and locations. A value abstraction (respectively a location abstraction) represents the possible values of an expression (resp. the possible locations pointed to an address) in the set of concrete states represented by the abstract domains. They implement an abstract semantics of the operations on expressions and addresses, except for dereferences that involve concrete states. To support further interactions between domains, value and location abstractions are extensible and can be combined into a standard reduced product. A typical example is the reduced product of intervals and congruences.

In `clike`, the evaluation of an expression or address either produces a concrete value, or fails when it exhibits an undefined behavior according to the C standard. The eventuality of such failures are represented by *alarms*, emitted by the abstract interpreter to report the undesirable behaviors of a program. They are collaboratively produced by all abstractions of values, locations or states.

Alarms, value and location abstractions are the communication interface between abstract domains. This chapter formalizes the alarms used by [EVA](#), and defines the requirements that the value (and location) abstractions must fulfil. It finally presents the value abstractions currently available in [EVA](#), and focuses on their representation and their abstract semantics of concrete values depending on the memory layouts (through integer arithmetic on variable addresses).

5.1 ALARMS

5.1.1 *Reporting Undesirable Behaviors*

Although abstract interpretation is a generic framework for inferring properties about programs, most static analyzers focus on proving the absence of *undesirable* behaviors. These behaviors are usually those that may lead to an error during a program execution, or to an unexpected result due to undefined behaviors in the C standard. [EVA](#) is no exception, and currently detects the following undefined behaviors (some of which are out of the scope of the simplified `clike` language):

- for lvalue dereferences: invalid memory accesses (NULL pointer dereferencing or out-of-bounds accesses); reading uninitialized

memory or dangling pointers (i.e. to out of scope variables);
writing in const memory;

- for arithmetic operations: divisions by zero, integer overflows, undefined bit shifts;
- for floating-point operations: infinite values or NaN results¹; overflows in the conversion from floating-point to integer;
- for pointer-related operations: subtractions between pointers ranging over different objects; function calls through a pointer with an incompatible type; invalid pointer comparisons;
- multiple side-effects on the same variable without a sequence point; assignments of an aggregate where the left- and right-hand-side overlap.

One of the main purposes of [EVA](#) is to emit an *alarm* at each program point where it fails to prove the absence of one of these undesirable behaviors. Each alarm may either reveal a real bug in the program, or be due to the over-approximations made through the abstractions used for the analysis. The latter alarms are *false* alarms. On the other hand, the soundness of the analysis guarantees that each *unsafe* statement, where an undesirable behavior may occur, is indeed annotated by an alarm. Thus, the alarms are themselves a sound over-approximation of the undesirable behaviors of the program. If an analysis raises no alarm on a program, then this program is free of the undesirable behaviors that [EVA](#) detects.

By pointing out all the potential runtime errors of a program, the alarms are the main result of the analyzer for the end user. We naturally aim at producing as few alarms as possible while remaining sound – ideally none on a program without any undesirable behavior. It is thus essential that the different abstractions used for the analysis may directly influence which alarms are generated. Therefore, alarms are part of the communication interface between the abstract domains and the analyzer.

5.1.2 Alarms as ACSL Assertions for the End User

The alarms that [EVA](#) emits are standard [ACSL](#) assertions [[ACSL](#)], handled by the FRAMA-C kernel. Each one is a guard against an undesirable behavior that [EVA](#) has failed to exclude. If such an assertion is satisfied for all possible executions of the program, then the corresponding undesirable behavior never occurs, meaning that the alarm was a false alarm. Otherwise, there is at least one execution that

¹ These are not undefined behaviors w.r.t. the ISO C99 or IEEE 754 specifications, but we choose to report them as undesirable errors.

violates the assertion, and this execution triggers the reported undesirable behavior.

To each assertion is associated a logical status, ranging over `true`, `false` and `unknown`. A `true` status is naturally given to the assertions that have been proven correct, which excludes the undesirable behavior. Otherwise, the undesirable behavior may occur (`unknown` status) or definitely happens if the program point is reachable in at least one concrete execution (`false` status). Thus, an assertion with a `false` status is *not* a false alarm (but it can be in dead code, and thus never triggered at runtime). The assertions are inserted into the source code, and are searchable in the GUI. In order not to overload the interface, EVA does not emit assertions it has already proven correct (i.e. with a `true` status): it does not report on the undesirable behaviors it has been able to exclude. To show them, it is possible to run the analysis with no abstract domain at all, to force the emission of the alarms for all possible undesirable behaviors our analyzer handles, and compare the outcome with the normal analysis.

At the end of an analysis, the conjunction of all emitted assertions is a sufficient condition for the code to be correct. An alarm with a `false` status precludes this, unless the code it concerns is dead. Otherwise, the assertions with an `unknown` status remain to be proven. If one can prove, one way or another, that they hold for all possible executions of the program, then they were all false alarms: the program is free of undesirable behaviors, even if our analyzer alone was unable to ensure it. To spare the burden of the manual proof of these assertions, the other plugins of FRAMA-C can also be used to prove them. The FRAMA-C kernel then ensures the automatic consolidation of the logical statuses set by different plugins on an assertion. Thus, a `true` status may eventually be set on all alarms issued by EVA.

5.1.3 Set of Possible Alarms

Alarms stem from illegal operations on expressions. They are produced by the abstract operators on values, accumulated during expression evaluation, and ultimately raised by the analyzer. In the meantime, the abstract domains may exclude some of them, thanks to the properties they have inferred. We define here the set of possible alarms that may be raised for a given expression.

5.1.3.1 Alarms from Operators

In the concrete semantics, the operators on expressions may return either a scalar value in $\bar{\mathbb{V}}$ in case of success, or an error Ω which denotes the undesirable behaviors that we track. For each operator, there is a finite set of logical conditions on its input values that are necessary and sufficient to exclude the error cases. To each one corresponds an ACSL assertion [ACSL, Figure 2.2], whose free variables are the

arguments of the operator. We denote by $\text{alarms}(\diamond)$ the set of these assertions: they are the possible alarms for the operator \diamond . As a reminder, $\llbracket \diamond \rrbracket(\vec{v})$ denotes the application of the operator \diamond on a vector \vec{v} of \mathbb{C} values. This operation fails if and only if at least one assertion of $\text{alarms}(\diamond)$ is not satisfied for the values \vec{v} of the arguments.

Definition 33. For an n -ary operator \diamond , $\text{alarms}(\diamond)$ is a set of assertions in $\overline{\mathbb{V}}^n \rightarrow \{\text{true}, \text{false}\}$ such that:

$$\overline{\llbracket \diamond \rrbracket}(\vec{v}) = \Omega \Leftrightarrow \exists A \in \text{alarms}(\diamond), \neg A(\vec{v})$$

As an example, the possible alarms for a signed integer addition are the arithmetic conditions avoiding integer overflows, which are on a 32-bits hardware:

$$\text{alarms}(\cdot + \cdot)(x, y) = \left\{ \begin{array}{l} -2147483648 \leq x + y, \\ x + y \leq 2147483647 \end{array} \right\}$$

And the possible alarms for a division prevent the divisor from being zero, and the quotient from overflowing, which can happen for $x = -2^{31}$ and $y = -1$ on the same hardware.

$$\text{alarms}(\cdot / \cdot)(x, y) = \left\{ \begin{array}{l} y \neq 0, \\ x / y \leq 2147483647 \end{array} \right\}$$

It is worth noting that the arithmetical operations used in these logical assertions are the mathematical operators with infinite precision.

Section 4.3.3 defined the concrete values as functions from memory layouts to \mathbb{C} values. Also, the semantics $\overline{\llbracket \diamond \rrbracket}^\theta$ of an operator on concrete values \vec{V} are its pointwise application on the \mathbb{C} values $\vec{V}(\theta)$ for each memory layout θ . Such semantics fail if and only if there exists one memory layout θ in which the operation on the \mathbb{C} values fails, that is if an alarm is not satisfied for $\vec{V}(\theta)$.

$$\forall \vec{V} \in \mathcal{V}^n,$$

$$\overline{\llbracket \diamond \rrbracket}^\theta(\vec{V}) \neq \Omega \Leftrightarrow \forall A \in \text{alarms}(\diamond), \forall \theta \in \Theta_P, A(\vec{V}(\theta))$$

Then, in a concrete state S , an operation $\diamond(e_1, \dots, e_n)$ succeeds if and only if for all layout θ , all the alarms for \diamond hold for the values of e_1 to e_n in the memory $S(\theta)$. For an alarm A , we write $A(e_1, \dots, e_n)(S)$ for the truth value of A applied to the values of e_1 to e_n in the memories $S(\theta)$ for all layouts θ .

Definition 34. Let \diamond be an n -ary operator and $A : \overline{\mathbb{V}}^n \rightarrow \{\text{true}, \text{false}\}$ an alarm of $\text{alarms}(\diamond)$. For any expressions e_1 to e_n , we define $A(e_1, \dots, e_n)$ as the assertion in $\mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ such that:

$$A(e_1, \dots, e_n)(S) \triangleq \forall \theta \in \Theta_P, A(\llbracket e_1 \rrbracket_{S(\theta)}, \dots, \llbracket e_n \rrbracket_{S(\theta)})$$

Lemma 4. For any n -ary operator \diamond and expressions e_1 to e_n , the operation $\diamond(e_1, \dots, e_n)$ succeeds in the concrete state S if and only if for any alarm A of $\text{alarms}(\diamond)$, the assertion $A(e_1, \dots, e_n)$ holds in S .

$$\overline{\llbracket \diamond(e_1, \dots, e_n) \rrbracket}^\Theta(S) \neq \Omega \Leftrightarrow \begin{cases} \forall A \in \text{alarms}(\diamond), A(e_1, \dots, e_n)(S) \\ \forall i \in \{1, \dots, n\}, \overline{\llbracket e_i \rrbracket}^\Theta(S) \neq \Omega \end{cases}$$

Proof. This directly results from the semantics $\overline{\llbracket \diamond(\vec{e}_i) \rrbracket}^\Theta(S)$ as stated in lemma 3. \square

5.1.3.2 Alarms from Dereferences

For a dereference, three alarms ensure respectively the validity of the memory read, and that the pointed value in memory is initialized and not a dangling address. Their validity depends on the memories of a state, and is thus defined through the layout-wise semantics of expressions, in a given concrete state S :

- $\backslash\text{valid_read}(a)(S)$ holds whenever the address a evaluates to the same valid τ -location l in every memory of S ;
- $\backslash\text{initialized}(a)(S)$ holds whenever no memory of S contains the `uninit` value at a byte of the τ -location of a ;
- $\neg\backslash\text{dangling}(a)(S)$ holds whenever no memory of S contains the `none` value at a byte of the τ -location of a .

$$\begin{aligned} \text{alarms}(*_\tau \cdot)(a) &\triangleq \{ \backslash\text{valid_read}(a); \\ &\quad \backslash\text{initialized}(a); \neg\backslash\text{dangling}(a) \} \\ \overline{\llbracket *_\tau a \rrbracket}^\Theta(S) \neq \Omega &\Leftrightarrow \begin{cases} \backslash\text{valid_read}(a)(S) \\ \wedge \backslash\text{initialized}(a)(S) \\ \wedge \neg\backslash\text{dangling}(a)(S) \end{cases} \end{aligned}$$

5.1.3.3 Alarms on Complete Expressions

The set of possible alarms is easily extended to any expression e , by gathering the union of all possible alarms of its operators and dereferences. Then, the evaluation of e in a concrete state S succeeds if and only if the assertions of $\text{alarms}(e)$ are satisfied for the values of the variables of e in all memories of the state.

Definition 35. For any expression e , the set $\text{alarms}(e)$ of possible alarms for this expression is the set of assertions in $\mathcal{S} \rightarrow \{\text{true}, \text{false}\}$ defined as:

$$\begin{aligned} \text{alarms}(\diamond(e_1, \dots, e_n)) &\triangleq \text{alarms}(\diamond)(e_1, \dots, e_n) \cup \left(\bigcup_{1 \leq i \leq n} \text{alarms}(e_i) \right) \\ \text{alarms}(*_\tau a) &\triangleq \text{alarms}(*_\tau \cdot)(a) \cup \text{alarms}(a) \end{aligned}$$

Lemma 5. *The evaluation of an expression e succeeds in a concrete state S if and only if all the assertions in $\text{alarms}(e)$ hold in S .*

$$\llbracket e \rrbracket^\Theta(S) \neq \Omega \Leftrightarrow \forall A \in \text{alarms}(e), A(S)$$

Proof. Immediate from the definition of the expression semantics from Figure 4.8: an expression fails if one of its operation fails (including dereferences). \square

These possible alarms are syntactically deduced from the expressions involved in the program statements. In the absence of abstract domains able to infer properties about the program behaviors, the analyzer would emit all of them. Otherwise, the abstractions used by the analyzer should be able to express a precise status for at least some of these alarms.

5.1.4 Maps of Alarms

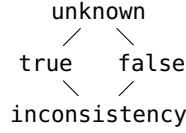
EVA aims at establishing whether undesirable behaviors may occur in a program execution. Proving the absence of undesirable behaviors amounts to prove the assertions of $\text{alarms}(e)$, for each expression e of the program. On the other hand, if an assertion is unsatisfied, then its undesirable behavior occurs in some execution. Thus, **EVA** strives to give a precise logical status to these assertions, during the evaluation of expressions.

5.1.4.1 Definition

Formally, the alarms propagated by **EVA** are partial maps from logical assertions to property statuses (Figure 5.1). An alarm map for the expression e must give a status to each assertion in $\text{alarms}(e)$, but does not necessarily contain explicitly all of these assertions. The missing assertions are implicitly bound to a default status. The set of assertions explicitly bound in a map \mathbf{A} is denoted $\text{dom}(\mathbf{A})$. **EVA** uses two different kinds of alarm maps, closed or open, with a different default status:

- closed maps bind missing assertions to `true`, excluding the undesirable behaviors they stand for. Thus, a closed map reports all the alarms that may occur for the given expression.
- open maps bind missing assertions to `unknown`. An open map gives a precise status to some alarms, but without guaranteeing completeness: all other alarms may occur. However, open maps are useful to assert the absence of some particular behaviors, without a complete knowledge of the evaluation.

Definition 36. An alarm map is a map from assertions to logical statuses. A closed map implicitly binds missing assertions to `true`, while an open map implicitly binds missing assertions to `false`.



(a) Lattice of statuses

$$\begin{aligned}
 \text{kind} &= \text{closed} \mid \text{open} \\
 \mathbb{A} &= (\text{assertion} \rightarrow \text{status}) \times \text{kind}
 \end{aligned}$$

(b) Alarm maps

$$\begin{aligned}
 \forall \mathbf{A} \in \mathbb{A}, \quad \text{default}(\mathbf{A}) &= \begin{cases} \text{true} & \text{if } \text{snd}(\mathbf{A}) = \text{closed} \\ \text{unknown} & \text{if } \text{snd}(\mathbf{A}) = \text{open} \end{cases} \\
 \mathbf{A}(a) &= \begin{cases} \mathbf{A}(a) & \text{if } a \in \text{dom}(\text{fst}(\mathbf{A})) \\ \text{default}(\mathbf{A}) & \text{otherwise} \end{cases}
 \end{aligned}$$

(c) Statuses bound to assertions in alarm maps

Figure 5.1: Statuses and alarms

Notation 4. The set of (open and closed) alarm maps is denoted \mathbb{A} .

As all the possible alarms for an expression are syntactically known, these partial maps are not absolutely necessary. In particular, an open map \mathbf{A} about an expression e can always be completed into a closed map by binding all its missing assertions from $\text{alarms}(e)$ to the unknown status. However, they make simpler the use of alarm maps as means of communication between abstractions. Value and state abstractions are thus simpler to write, because they can focus on the alarms they are able to understand. Also, adding new alarms can be done in a transparent way.

5.1.4.2 Semantics of Alarm Maps

We define the semantics of alarm maps as abstractions of the undesirable behaviors that arise from the application of an n-ary operator \diamond to a vector \vec{V} of concrete values.

Definition 37. An alarm map \mathbf{A} is a sound abstraction of an operation $\overline{[\diamond]}(\vec{V})$ if and only if:

- all alarms of \mathbf{A} are alarms of $\text{alarms}(\diamond)$;
- the precise statuses assigned by \mathbf{A} to the alarms in $\text{alarms}(\diamond)$ are correct for the values \vec{V} .

We denote this fact by $\mathbf{A} \models_{\mathbb{A}} \overline{[\diamond]}(\vec{V})$.

$$\begin{aligned}
& \forall \vec{V} \in \overline{V}^n, \quad \forall \mathbf{A} \in \mathbb{A}, \\
& \mathbf{A} \models_{\mathbb{A}} \overline{[\diamond]}(\vec{V}) \Leftrightarrow \\
& \quad \begin{cases} \forall A \in \mathbf{A}, \quad A \in \text{alarms}(\diamond) \\ \forall A \in \text{alarms}(\diamond), \mathbf{A}(A) = \text{true} \Rightarrow \forall \theta \in \Theta_P, A(\vec{V}(\theta)) \\ \forall A \in \text{alarms}(\diamond), \mathbf{A}(A) = \text{false} \Rightarrow \forall \theta \in \Theta_P, \neg A(\vec{V}(\theta)) \end{cases}
\end{aligned}$$

Firstly, this definition requires that the assertions carried by the alarm maps indeed prevent undesirable behaviors of the operation. An alarm map cannot contain any assertion unrelated to the given operation. Secondly, it also requires that the precise statuses assigned to assertions in \mathbf{A} are correct in all memory layouts: the assertions bound to true (resp. false) in A are satisfied (resp. their negation is satisfied) on the C values $\vec{V}(\theta)$, for any layout θ . This condition is extended to the omitted assertions of closed maps, that are implicitly bound to true. (As the default status of open map is imprecise, the condition does not apply to the missing assertions of open maps.)

For a closed map, as the missing assertions are bound to true, the conjunction of all these assertions ensures the absence of undesirable behavior: if all the assertions are satisfied for the values $\vec{V}(\theta)$ in all memory layouts θ , then the computation of $\diamond(\vec{V})$ succeeds.

Lemma 6. *If a closed map \mathbf{A} is a sound abstraction of the operation $\overline{[\diamond]}^{\Theta}(\vec{V})$, then the assertions in \mathbf{A} ensure that the computation succeeds.*

$$\begin{cases} \mathbf{A} \models_{\mathbb{A}} \overline{[\diamond]}(\vec{V}) \\ \mathbf{A} \text{ closed} \end{cases} \Rightarrow \forall \theta \in \Theta_P, \forall A \in \mathbf{A}, A(\vec{V}(\theta)) \Rightarrow \overline{[\diamond]}(\vec{V}) \neq \Omega$$

Proof. Let \mathbf{A} be such a map. By definition of closed map, for any alarm $A \in \text{alarms}(\diamond)$, either $A \in \mathbf{A}$ or $\mathbf{A}(A) = \text{true}$. If for all alarms $A \in \mathbf{A}$, we have $\forall \theta \in \Theta_P, A(\vec{V}(\theta))$, then we have this property for all alarms of $\text{alarms}(\diamond)$, as $\mathbf{A}(A) = \text{true}$ also implies it. Thus, definition 33 ensures $\overline{[\diamond]}(\vec{V}) \neq \Omega$. \square

The soundness definition of alarm maps is similarly extended to the evaluation of any expression e in a concrete state S . The evaluation of an expression fails if one of the involved operators fails. A map of alarms \mathbf{A} is a sound abstraction of the evaluation $\overline{[e]}^{\Theta}(S)$ if each assertion of \mathbf{A} belongs to $\text{alarms}(e)$, and if the statuses of A are correct in the state S . Once again, this latter condition also applies to the assertion of $\text{alarms}(e)$ implicitly bound to the default status of the map. If the map is closed, then the evaluation of e succeeds in this state if all its assertions are satisfied.

Definition 38. An alarm map \mathbf{A} is a sound abstraction of the evaluation of an expression e in a concrete state S if and only if:

- the alarms of \mathbf{A} are alarms of $\text{alarms}(e)$;
- the precise statuses assigned by \mathbf{A} to the alarms in $\text{alarms}(e)$ are correct in the state S .

We denote this fact by $\mathbf{A} \models_{\mathbb{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S)$.

$$\begin{aligned} \forall e \in \text{expr}, \quad \forall S \in \mathcal{S}, \quad \forall \mathbf{A} \in \mathbb{A}, \\ \mathbf{A} \models_{\mathbb{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S) \Leftrightarrow \\ \left\{ \begin{array}{l} \forall A \in \mathbf{A}, \quad A \in \text{alarms}(e) \\ \forall A \in \text{alarms}(e), \quad \mathbf{A}(A) = \text{true} \Rightarrow A(S) \\ \forall A \in \text{alarms}(e), \quad \mathbf{A}(A) = \text{false} \Rightarrow \neg A(S) \end{array} \right. \end{aligned}$$

Lemma 7. *If a closed map \mathbf{A} is a sound abstraction of the evaluation $\overline{\llbracket e \rrbracket}^{\Theta}(S)$, then the assertions in \mathbf{A} ensure that the computation succeeds.*

$$\left. \begin{array}{l} \mathbf{A} \models_{\mathbb{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S) \\ \mathbf{A} \text{ closed} \end{array} \right\} \Rightarrow \forall \theta \in \Theta_P, \forall A \in \mathbf{A}, A(S) \Rightarrow \overline{\llbracket e \rrbracket}^{\Theta}(S) \neq \Omega$$

Proof. Same reasoning than for lemma 6, by using lemma 5. \square

As a consequence, if the empty closed map is a sound abstraction of an evaluation, then the evaluation always succeeds.

5.1.4.3 Lattice Structure of Alarm Maps

The alarms are also equipped with a bounded lattice structure. The join $\sqcup_{\mathbb{A}}$ and meet $\sqcap_{\mathbb{A}}$ are defined pointwise. The domains of the two maps are equalized by adding in each map the assertions present only in the other, bound to the default status of the map. Then, the join or meet of the statuses lattice (Figure 5.1) is applied pointwise on the maps, as well as on their default status. Thus, the join or meet of two maps of the same kind returns a map of this kind; the join with an open map returns an open map, while the meet with a closed map returns a closed map. The meet may discover an inconsistency between statuses, which stops the analysis as it means the analysis is incorrect. The bottom of the alarms lattice is the closed empty map, denoting an absence of undesirable behavior. Its top is the open empty map: any undesirable behavior may happen, and no assertion has a precise status (i.e. they all have implicitly the unknown status).

Lemma 8. *If \mathbf{A}_1 and \mathbf{A}_2 are both sound abstractions of the evaluation $\overline{\llbracket e \rrbracket}^{\Theta}(S)$, then the meet $\mathbf{A}_1 \sqcap_{\mathbb{A}} \mathbf{A}_2$ is a sound abstraction of the evaluation $\overline{\llbracket e \rrbracket}^{\Theta}(S)$.*

$$\left. \begin{array}{l} \mathbf{A}_1 \models_{\mathbb{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S) \\ \mathbf{A}_2 \models_{\mathbb{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S) \end{array} \right\} \Rightarrow (\mathbf{A}_1 \sqcap_{\mathbb{A}} \mathbf{A}_2) \models_{\mathbb{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S)$$

Proof. Let e be an expression, S a concrete state, and \mathbf{A}_1 and \mathbf{A}_2 two sound abstractions of the evaluation $\overline{\llbracket e \rrbracket}^\Theta(S)$. The alarms carried by \mathbf{A}_1 or \mathbf{A}_2 are in $\text{alarms}(e)$; an alarm of the meet is an alarm from \mathbf{A}_1 or \mathbf{A}_2 , and thus is in $\text{alarms}(e)$. The status assigned by each map to the alarm is correct, and thus the meet of theses statuses—which is the status of the meet of the maps—is also correct. \square

5.1.5 Propagating Alarms and Bottom Elements

Each operator on expressions yields a value or raises an undesirable behavior. The abstract semantics take into account both possibilities. Thus, each abstract operator produces an abstraction of the possible resulting value and an alarm map. During the evaluation of an expression e , the intermediate value abstractions are directly consumed by further operators, until finally reaching a value abstraction for e . On the other hand, the alarm maps produced at each step of an evaluation are joined together. Indeed, the final alarm map must over-approximate the possible undesirable behaviors of all the operators involved in the evaluation of e . Thus, a sound abstraction of the undesirable behavior of the evaluation of the expression $\diamond(e_1, \dots, e_n)$ is the join of:

- the alarm maps for the complete evaluation of each operand e_i
- the alarm maps for the application of the operator \diamond to the values of its operands.

Lemma 9. Let \diamond be an n -ary operator, e_1 to e_n be n expressions and S be a concrete state. Let n alarm maps \mathbf{A}_1 to \mathbf{A}_n be respectively sound abstractions of the evaluation of the expressions e_1 to e_n in S . Let an alarm map \mathbf{A}_\diamond be a sound abstraction of the operation $\overline{\llbracket \diamond \rrbracket}(\vec{V})$, where \vec{V} is the vector of concrete values of the expressions e_1 to e_n in S . The join between $\mathbf{A}_1, \dots, \mathbf{A}_n$ and \mathbf{A}_\diamond is a sound abstraction of the evaluation of the expression $\diamond(e_1, \dots, e_n)$ in the state S .

$$\begin{aligned}
 & \forall i \in \{1, \dots, n\}, \mathbf{A}_i \models_{\mathbb{A}} \overline{\llbracket e_i \rrbracket}^\Theta(S) \\
 & \wedge \forall \theta \in \Theta_P, \mathbf{A}_\diamond \models_{\mathbb{A}} \overline{\llbracket \diamond \rrbracket}(\overline{\llbracket e_1 \rrbracket}_{S(\theta)}, \dots, \overline{\llbracket e_n \rrbracket}_{S(\theta)}) \\
 & \quad \Downarrow \\
 & (\bigsqcup_{1 \leq i \leq n} \mathbf{A}_i) \sqcup_{\mathbb{A}} \mathbf{A}_\diamond \models_{\mathbb{A}} \overline{\llbracket \diamond(e_1, \dots, e_n) \rrbracket}^\Theta(S)
 \end{aligned}$$

Lemma 10. Let a be an address, and S be a concrete state. If the alarm map \mathbf{A} is a sound abstraction of the evaluation of the address a and \mathbf{A}_* is a sound abstraction of the dereference $\ast(\overline{\llbracket a \rrbracket}^\Theta(S))$, then the join between \mathbf{A} and \mathbf{A}_* is a sound abstraction of the evaluation $\overline{\llbracket \ast_\tau a \rrbracket}^\Theta(S)$.

$$\left. \begin{array}{l} \mathbf{A} \models_{\mathbb{A}} \overline{\llbracket a \rrbracket}^\Theta(S) \\ \mathbf{A}_* \models_{\mathbb{A}} \overline{\llbracket \ast \llbracket a \rrbracket \rrbracket}^\Theta(S) \end{array} \right\} \Rightarrow \mathbf{A} \sqcup_{\mathbb{A}} \mathbf{A}_* \models_{\mathbb{A}} \overline{\llbracket \ast_\tau a \rrbracket}^\Theta(S)$$

```

1 type 'a or_bottom = [ 'Value of 'a | 'Bottom ]
  type 't with_alarms = 't * Alarmset.t
  type 't evaluated = 't or_bottom with_alarms

5 let (>=>) (t, a) f =
    match t with
    | 'Bottom -> 'Bottom, a
    | 'Value t -> let t', a' = f t in t', Alarmset.union a a'

```

Figure 5.2: OCaml types for the alarms and the bottom case

Proof. Both lemmas stem directly from definition 35 of the set of possible alarms for an arbitrary expression. \square

An alarm gets a false status when the analysis guarantees that its undesirable behavior always occurs, meaning that the evaluation of the given expression always fails. An alarm with a false status should always be issued with the special value abstraction \perp , which means that no value can be produced. This canonical value is used to complement the lattice structure of each abstraction used by our analyzer. It denotes an abstraction with an empty concretization, i.e. an unreachable state, and is shared between all abstractions. When this bottom value \perp arises for the evaluation of a subterm, then the complete evaluation fails and returns \perp as well.

Figure 5.2 presents the OCaml type related to the alarms and the bottom element. The bottom case is implemented via a polymorphic variant, to simplify its use through the analyzer. The types `or_bottom` and `with_alarms` complement any existing type respectively with the bottom case and with a map of alarms. The type `evaluated` is the combination of both, and is pervasive in the implementation of the forward abstract semantics of expression. During an evaluation, we use the `>=>` monad to join the alarms produced at each step, and to propagate the bottom case when it arises.

5.2 ABSTRACTIONS OF CONCRETE VALUES

The alarms over-approximate the *undesirable* behaviors of a program. The other abstractions over-approximate the *correct* behaviors of the program, and do not consider the erroneous cases that are taken into account through the alarms. These other abstractions are divided into several layers, each operating at a different level of the program semantics. The bottom layer is made of over-approximations of the C scalar value an expression may have at a given program point. Those approximations are abstractions of values of integer types, real floating-point types or pointer type. They model the semantics of expressions, and are part of the communication interface between the

different abstractions of the top layer, which models the semantics of statements.

This section first defines the soundness of such value abstractions. It then presents their lattice structure and their abstraction of the semantics of constants and operators. It finally outlines the concept of location abstractions, that similarly abstract the semantics of addresses.

5.2.1 Concretization and Soundness of Value Abstractions

Values in $\mathbb{V}^\#$ are abstractions of sets of concrete values of scalar types. They implement an abstract semantics of expressions. The soundness of these abstract semantics is defined through a concretization function γ_v , that connects each value to the set of concrete values it represents. As explained in Section 4.3, while the C value of an expression may depend on the memory layout, our concrete semantics should not. Let us remind that in this semantics, a concrete value is a C value for each correct memory layout of the given program P , represented as a function from memory layouts in Θ_P to scalar values in $\overline{\mathbb{V}}$. The set of concrete values is written \mathcal{V} .

$$\gamma_v : \mathbb{V}^\# \rightarrow \mathcal{P}(\mathcal{V}) \quad \text{with } \mathcal{V} \triangleq \Theta_P \rightarrow \overline{\mathbb{V}}$$

In order to illustrate this approximation of functions, we introduce here two kinds of integer value abstractions based on intervals. We will use them as example in this section.

- Basic intervals, denoted as $[i..s]$, represent sets of scalar values independent from the memory layout, that are constant functions in our concrete semantics. More formally, the concretization of a basic interval is defined as:

$$\gamma_v([i..s]) = \{\lambda_. v \mid i \leq v \leq s\}$$

- Garbled intervals, denoted as $[i..s]^{\Theta_P}$, may represent values that depend on the memory layout. Their concretization includes any function whose image is in the given interval.

$$\gamma_v([i..s]^{\Theta_P}) = \{f \mid \forall \theta \in \Theta_P, i \leq f(\theta) \leq s\}$$

An expression independent from the memory layout has the same C value for all of them. A precise abstraction of its possible values may have a concretization made of constant functions only, thus encoding this independence. If the expression is a constant, a precise abstraction of its value represents a single constant function. Less precise abstractions could however fail to maintain these properties.

Example 8. Consider the code sample of Figure 5.3. At the end of the main function, the variables i and j have an integer value between 0

```

1 void main (int c) {
    unsigned int i = c ? 0 : 10;
    unsigned int j = 10 / ((unsigned int) &i)
}

```

Figure 5.3: Value abstraction of some expressions

and 10, but the value of j depends on the memory layout through the address of i . In the concrete semantics, the concrete value of i is a constant function among $\lambda_.0$ and $\lambda_.10$, according to the value of the main argument c . The most precise interval abstraction of these functions is the basic interval $[0..10]$. Any larger basic interval is also a sound approximation of the concrete values of i , although less precise. These basic intervals not only encode a range for the integer value of the variable i , but also the fact that these values are independent from the memory layout.

On the other hand, the garbled interval $[0..10]^{\Theta_P}$ (or any larger garbled interval) is also a sound approximation of the concrete values of i , but fails to encode this fact. This garbled interval is also a sound approximation of the set of possible concrete values of j , that cannot be represented by a basic interval.

Value abstractions represent function from memory layouts to C values. To simplify this semantics, we may be tempted to consider the set of C values denoted by an abstraction $v^\#$ in a given memory layout θ : it is the union of the image of θ by all functions in the concretization of $v^\#$. We denote this set by $\gamma_v(v^\#)(\theta)$, even if $\gamma_v(v^\#)$ is a set of functions.

$$\gamma_v(v^\#)(\theta) = \{f(\theta) \mid f \in \gamma_v(v^\#)\}$$

This view of a value abstraction yields a set of C values in each memory layout of Θ_P , but leaves out some information given by the set of functions. For instance, an abstraction can encode that the value of an expression is independent of the memory layout, even without any information about its precise value. This is the case of the basic interval $[-\infty..\infty]$. Its concretization is then the set of constant functions $\lambda_.k$ in \mathcal{V} , much more precise than the set of all these functions. However, for all memory layouts, the set $\gamma_v([-\infty..\infty])(\theta)$ is the set of all values $\overline{\mathbb{V}}$, and the property carried by the abstraction is lost. Yet, this property may be important if the expression is used as a conditional in an if statement, as the control flow of a program should not depend on the memory layout.

Still, in a memory layout θ , an abstraction $v^\#$ is a *sound* approximation of the value of an expression e in a given concrete memory m

when e evaluates in the memory m into a C value that belongs to the concretization $\gamma_v(v^\#)(\theta)$:

$$\exists f \in \gamma_v(v^\#), \llbracket e \rrbracket_m = f(\theta) \Leftrightarrow \llbracket e \rrbracket_m \in \gamma_v(v^\#)(\theta)$$

However, according to a concrete state, the soundness of a value abstraction is not defined for each memory layout independently. The concrete states of our language semantics are functions from correct memory layouts to memories. The evaluation of a scalar expression in a concrete state is then a function from memory layouts to scalar values. The soundness of value abstraction is defined accordingly.

Definition 39. An abstraction $v^\#$ is a *sound* approximation of the value of an expression e in a given concrete state S when the evaluation of e in S is in the concretization of $v^\#$.

$$\llbracket e \rrbracket(S) \in \gamma_v(v^\#)$$

5.2.2 Lattice Structure

A value abstraction $\mathbb{V}^\#$ must be equipped with a lattice structure $(\mathbb{V}^\#, \sqsubseteq_v, \sqcup_v, \sqcap_v, \top_v)$ that satisfies the properties specified in Figure 2.4, according to the value concretization γ_v . The join and the meet over-approximate respectively the union and the intersection of sets of concrete values. Value abstractions are partially ordered according to the inclusion of their concretization. The top element represents all possible C values of scalar type.

During the analysis, two values are met when they are both sound abstractions of the possible values of the same lvalue or expression. Then, the meet produces the most precise value abstraction for this expression. In particular, each abstract domain supplies the analyzer with value abstractions during the evaluation of an expression. The generic evaluator performs naturally the meet of these values, to benefit from the information inferred by all domains.

By contrast with the lattice structure required for state abstractions, the join between value abstractions is not really crucial. In abstract interpretation, the join of abstract states is used at a merge point in the CFG, to over-approximate the concrete states coming from each branch. But there is not such thing in our expressions (as conditional operators are compiled into conditionals statements in CIL). In EVA, the join on values is only used when the evaluation of an expression is separated into several cases, to rejoin the abstractions computed for each case. The process of subdividing an evaluation is explained in Section 6.2.4.

The state abstractions and the generic analyzer naturally resort to the top element of the value lattice whenever they have no information about a variable or an expression.

5.2.3 Semantics of Values and Alarms

An alarm map models the undesirable behaviors that may occur when the evaluation of an expression fails into the error value Ω . If the assertions carried by such a map are satisfied at runtime, then the evaluation cannot fail. A value abstraction approximates the concrete value an expression may have when its evaluation does not fail. A pair of a value abstraction and an alarm map can thus exhaustively represent the evaluation of an expression.

Definition 40 (Semantics of pairs of values and alarms). A pair of a value abstraction v and an alarm map \mathbf{A} is a sound abstraction of the evaluation of an expression e in a concrete state S when \mathbf{A} is a sound approximation of the undesirable behaviors of the evaluation, and v is a sound approximation of its resulting value. We denote that by $(v, \mathbf{A}) \models_{\mathbb{V} \times \mathbb{A}} \llbracket e \rrbracket^\Theta(S)$.

$$(v, \mathbf{A}) \models_{\mathbb{V} \times \mathbb{A}} \llbracket e \rrbracket^\Theta(S) \Leftrightarrow \begin{cases} \mathbf{A} \models_{\mathbb{A}} \llbracket e \rrbracket^\Theta(S) \\ \llbracket e \rrbracket^\Theta(S) \subseteq \gamma_{\mathbb{V}}(v) \cup \{\Omega\} \end{cases}$$

We also define the lattice structure of such pairs, by the pointwise application of the lattice structure of value abstractions and alarm maps.

Definition 41 (Lattice structure of pairs of values and alarms).

$$\begin{aligned} \forall (v_1, v_2) \in (\mathbb{V}^\#)^2, \forall (\mathbf{A}_1, \mathbf{A}_2) \in \mathbb{A}^2, \\ (v_1, \mathbf{A}_1) \sqsubseteq_{\mathbb{V} \times \mathbb{A}} (v_2, \mathbf{A}_2) &\Leftrightarrow v_1 \sqsubseteq_{\mathbb{V}} v_2 \wedge \mathbf{A}_1 \sqsubseteq_{\mathbb{A}} \mathbf{A}_2 \\ (v_1, \mathbf{A}_1) \sqcup_{\mathbb{V} \times \mathbb{A}} (v_2, \mathbf{A}_2) &\triangleq (v_1 \sqcup_{\mathbb{V}} v_2, \mathbf{A}_1 \sqcup_{\mathbb{A}} \mathbf{A}_2) \\ (v_1, \mathbf{A}_1) \sqcap_{\mathbb{V} \times \mathbb{A}} (v_2, \mathbf{A}_2) &\triangleq (v_1 \sqcap_{\mathbb{V}} v_2, \mathbf{A}_1 \sqcap_{\mathbb{A}} \mathbf{A}_2) \\ \top_{\mathbb{V} \times \mathbb{A}} &\triangleq (\top_{\mathbb{V}}, \top_{\mathbb{A}}) \end{aligned}$$

Lemma 11. If two pairs of value abstractions and alarm maps (v_1, \mathbf{A}_1) and (v_2, \mathbf{A}_2) are sound abstractions of the evaluation of an expression e in a concrete state S , then their meet is also a sound abstraction of this evaluation $\llbracket e \rrbracket^\Theta(S)$.

$$\left. \begin{array}{l} (v_1, \mathbf{A}_1) \models_{\mathbb{V} \times \mathbb{A}} \llbracket e \rrbracket^\Theta(S) \\ (v_2, \mathbf{A}_2) \models_{\mathbb{V} \times \mathbb{A}} \llbracket e \rrbracket^\Theta(S) \end{array} \right\} \Rightarrow (v_1, \mathbf{A}_1) \sqcap_{\mathbb{V} \times \mathbb{A}} (v_2, \mathbf{A}_2) \models_{\mathbb{V} \times \mathbb{A}} \llbracket e \rrbracket^\Theta(S)$$

Proof. This lemma is a direct consequence of the properties of Figure 2.4 and of lemma 8. \square

5.2.4 Abstraction of Constants

Some lifting functions embed concrete values into abstract ones. They are used to translate the constants into the abstracted world. For a

constant c of the language, we write $c^\#$ its injection in the value abstraction $\mathbb{V}^\#$: $c^\#$ is simply the smallest abstract element v of $\mathbb{V}^\#$ such that $c \in \gamma_v(v)$. Let us recall that the constants of our language include integer, floating-point and pointer values. For a value abstraction unable to represent some types, $c^\#$ could be \top_v for some constants c .

5.2.5 Abstraction of Operators

The main feature of value abstractions is to implement an abstract semantics for the C operators on expressions. For each n -ary operation \diamond on C expressions, the value abstraction provides a forward and a backward abstract counterpart $F_\diamond^\#$ and $B_\diamond^\#$. Their correctness is defined through the concretization γ_v . As the concretization of a value abstraction is a set of concrete values, we silently lift the semantics of operators from elements to sets. By convenience, we also write $\gamma_v(\vec{v})$ for the concretization of a vector \vec{v} of value abstractions.

Notation 5 (Concretization of a vector of abstractions, and concrete semantics on sets).

$$\begin{aligned} \forall \vec{v} = (v_1, \dots, v_n) \in (\mathbb{V}^\#)^n, \\ \gamma_v(\vec{v}) &\triangleq \gamma_v(v_1), \dots, \gamma_v(v_n) \\ \overline{[\diamond]}^\Theta(\gamma_v(\vec{v})) &\triangleq \{\overline{[\diamond]}^\Theta(\vec{V}) \mid \vec{V} \in \gamma_v(\vec{v})\} \end{aligned}$$

5.2.5.1 Forward Abstract Semantics

Given abstractions \vec{v} of the operands, the forward function $F_\diamond^\#$ produces a sound abstraction of the output of the operation, when applied to concrete values in the concretization of \vec{v} . The resulting abstraction of $F_\diamond^\#$ consists of:

- an alarm map that is a sound abstraction of the undesirable behaviors of the evaluation of $\overline{[\diamond]}^\Theta(\gamma_v(\vec{v}))$.
- a value that is an over-approximation of the set of the possible concrete values resulting from $\overline{[\diamond]}^\Theta(\gamma_v(\vec{v}))$, when no undesirable behavior occurs. This value may be \perp , if the operation is completely illegal (i.e. if it always fails with the error Ω).

Definition 42. A forward abstract semantics of an n -ary operator \diamond is a function $F_\diamond^\#$ from a vector \vec{v} of n value abstractions to a pair of a value abstraction and an alarm map that are sound abstractions of the possible output of the operator, when applied to concrete values in the concretization of \vec{v} .

$$\begin{aligned} F_\diamond^\# : (\mathbb{V}^\#)^n &\rightarrow (\mathbb{V}^\# + \perp) \times \mathbb{A} \\ \forall \vec{v} \in (\mathbb{V}^\#)^n, \quad F_\diamond^\#(\vec{v}) = (r, \mathbf{A}) &\Rightarrow \begin{cases} \mathbf{A} \models_{\mathbb{A}} \overline{[\diamond]}^\Theta(\gamma_v(\vec{v})) \\ \overline{[\diamond]}^\Theta(\gamma_v(\vec{v})) \subseteq \gamma_v(r) \cup \{\Omega\} \end{cases} \end{aligned}$$

5.2.5.2 Backward Abstract Semantics

Conversely, the backward function $B_{\diamond}^{\#}$ produces abstractions of the operands from an abstraction of the result. However, such a backward propagation of abstractions cannot be precise for operators with several operands, as multiple abstractions should be deduced from only one. Indeed, knowing that $x \leq y$ or that $x + y \in [0..10]$ in all layouts is not enough to infer any information about the value of x and y . Instead, a backward operator $B_{\diamond}^{\#}$ takes the value abstractions of the operands, and tries to reduce them according to the abstraction of the result. The reduced abstractions must still over-approximate all the possible concrete values for the arguments leading to a value included in the result abstraction through the operator. For instance, the backward operation on interval abstractions for the comparison $\top \leq [0..10]$ with result $[1]$ reduces the first argument value to $[-\infty..10]$, and lets the second argument value unchanged. The backward addition $\top + [0..+\infty]$ with a result in $[0..10]$ leads to the same result.

Definition 43. A backward abstract semantics of an n -ary operator \diamond is a function $B_{\diamond}^{\#}$ that takes an abstraction r and a vector \vec{v} of n abstractions, and that produces a vector of value abstractions that are sound abstractions of the possible inputs in \vec{v} for which the operator leads to a result in the concretization of r .

$$B_{\diamond}^{\#} : (\mathbb{V}^{\#})^n \times \mathbb{V}^{\#} \rightarrow (\mathbb{V}^{\#} + \perp)^n$$

$$\forall \vec{v}, r \in (\mathbb{V}^{\#})^{n+1}, \quad \{\vec{x} \in \gamma_v(\vec{v}) \mid \overline{[\diamond]}^{\theta}(\vec{x}) \in \gamma_v(r)\} \subseteq \gamma_v(B_{\diamond}^{\#}(\vec{v}, r))$$

The backward operators handle only value abstractions, and do not produce or reduce alarms. Indeed, while the value of $\diamond(\vec{e})$ depends directly on the value of the expressions \vec{e} , the alarms issued for the operation are independent from those issued for its operands. Also, the backward operators can assume that the operation succeeds, since the alarms produced by the forward abstract semantics report the error cases. This may achieve some reductions of the operands, if some values lead necessarily to an undesirable behavior. However, backward operators may also reveal a contradiction, leading to the empty value abstraction \perp for one of the operand, and so for the whole expression. This happens typically when the condition of an `if` statement is unsatisfied.

5.2.5.3 Implementation Signature

Figure 5.4 presents the OCaml interface of the abstract operators of value abstractions in [EVA](#). The type `t` is the one of the value abstraction. Operators are grouped by arity; `unop` and `binop` are the unary and binary operators of the [CIL AST](#). The type on which this operator is applied is also explicitly given, as the [CIL](#) operators are not

```

1  type t (* type of value abstraction *)

    val forward_unop :
        context:unop_context -> typ -> unop -> t -> t evaluated
5  val forward_binop :
        context:binop_context -> typ -> binop -> t -> t -> t evaluated
    val forward_cast :
        context:exp -> src_typ:typ -> dst_typ: typ -> t -> t evaluated

10 val backward_unop :
        typ -> unop -> arg:t -> res:t -> t option or_bottom
    val backward_binop :
        typ -> binop -> left:t -> right:t -> result:t ->
        (t option * t option) or_bottom
15 val backward_cast:
        src_typ: typ -> dst_typ: typ -> src_val: t -> dst_val: t ->
        t option or_bottom

```

Figure 5.4: Interface of the abstract operators on values

specialized by their type. The context argument gathers the expressions of the operands; they are only needed to create the proper map of alarms for this particular application of the operator. Indeed, the relevant alarms are assertions whose free variables are these expressions. Finally, an abstract operator takes the value abstractions of the operands, and returns an abstraction of the result —value and alarms— through the `t evaluated` type introduced in Section 5.1.5.

Likewise, a backward operator takes a value abstraction for each operand and for the result, and returns either bottom —in case of contradiction— or a value abstraction for each operand. In many cases, the abstraction of the result is too imprecise to achieve a reduction of the operands. The backward operator could then return the operand abstractions unchanged, but it may be useful to notify explicitly the absence of reduction. To this end, the results of the backward operators are OCaml option types, `None` meaning that no reduction has been performed.

Similar functions are provided for conversions, which are not standard unary operators in CIL, and for which the source and destination types are given explicitly.

5.2.6 Abstractions of Memory Locations

Values are abstractions of C values of scalar types only. Copies of aggregate types such as arrays and structures must be handled by the state abstractions – although the content of their fields and cells can be represented as values. However, in a language with pointers such as C, scalar variables may contain addresses. Hence, values are also

abstractions of pointer values, and thus of memory addresses. The abstract (forward and backward) transformers on value abstractions encompass all *expression* operations involving pointers: addition of an integer to a pointer, subtraction of two pointers, comparison of pointers, conversion between integer types and pointers.

However, the abstract value operators do not include the address operations —array subscripting and structure fields. Instead, the address semantics is abstracted by specific representations of memory locations. They are abstractions of the set of the possible memory locations pointed to by an address in some concrete states. They implement an abstract semantics of addresses, and are used to process dereferencing through the state abstractions of the memory. These abstractions of address locations are closely tied to the abstractions of the expression values. Indeed, an address is an expression (evaluating to a pointer value, thus a location), plus certain offsets for fields in structures and cells in arrays, the latter being also denoted by expressions (evaluating to integers). In fact, the abstractions for locations could be exactly the same as for the pointer values. However, the distinction between the two allows to gain precision and some subtleties in the handling of addresses. In particular, [EVA](#) handles bitfields through the location abstractions only (although `clike` does not include bitfield).

It is worth remembering that in `clike`, addresses appear only in dereferences and assignments. In a given concrete state, the dereference and assignment semantics both require their address to evaluate to the same valid τ -location in all possible memory layouts. Thus, the location abstractions approximate sets of constant locations, identified as valid pointer values for the considered type. If the value of an address depends on the memory layout, an alarm is emitted.

A set of such location abstractions is denoted $\mathbb{L}^\#$. A concretization $\gamma_{\mathbb{L}}$ links each location abstraction to the set of memory locations it represents.

$$\gamma_{\mathbb{L}} : \mathbb{L}^\# \rightarrow \mathcal{P}(\mathcal{L})$$

As usual, location abstractions are equipped with a lattice structure $(\mathbb{L}^\#, \sqsubseteq_{\mathbb{L}}, \sqcup_{\mathbb{L}}, \sqcap_{\mathbb{L}}, \top_{\mathbb{L}})$. Finally, location abstractions provide the abstract address semantics:

- $F_{\text{addr}}^\#$ converts a value abstraction v into a location abstraction (as an expression with pointer type is also an address): the result only abstracts the constant valid pointer values of $\gamma_{\mathbb{V}}(v)$.
- $F_{[]}^\#$ abstracts array subscripting $a[e]$, from a location abstraction of the address a and a value abstraction of the expression e .
- $F_{.f}^\#$ abstracts the location of the field f in the structures represented by the given location abstraction.

Listing 5.1: Reducing abstract location

```

1 void main (int i) {
    int t[2] = {1,2};
    int x = t[i];
    int *p = i ? &x : NULL;
5   int y = *p;
    }

```

$$\begin{aligned}
 F_{\text{addr}}^{\#} &: \mathbb{V}^{\#} \rightarrow (\mathbb{L}^{\#} + \perp) \times \mathbb{A} \\
 F_{\square}^{\#} &: \mathbb{L}^{\#} \rightarrow \mathbb{V}^{\#} \rightarrow (\mathbb{L}^{\#} + \perp) \times \mathbb{A} \\
 F_{\cdot f}^{\#} &: \mathbb{L}^{\#} \rightarrow (\mathbb{L}^{\#} + \perp) \times \mathbb{A}
 \end{aligned}$$

These forward functions also have their backward counterparts. As for value operator \diamond , these operations do not depend on memories. Their abstract forward and backward semantics are guided by the same principles and the same soundness requirements as the semantics of expression. We do not detail them here, and refer the reader to Section 5.2.5.

5.2.6.1 Reductions on Addresses

As mentioned earlier, the dereference of an address forces some requirements about the value of the address, regardless of the concrete state and the resulting concrete value. In the absence of dynamic allocation, the validity of an address can be checked without any knowledge of the memory. A dereferenced address must have the same valid pointer value in all memory layout, i.e. a value such as $\lambda_{\cdot}(\&x, i)$ with x a variable of the program, and i an integer between 0 and $\text{sizeof}(x) - 1$. Otherwise, the dereference raises an error.

In EVA, the location semantics reflects this:

- if an abstract location does not guarantee the validity of the address, an alarm is emitted;
- if possible, abstract locations are shrunk to their valid parts for a dereference. This includes reducing the abstraction of an array index to a positive integer value, strictly less than the size of the array.

Example 9. The code of listing 5.1 includes two dereferences that fails for some initial value of the argument i . At line 3, the abstract semantics $F_{\square}^{\#}$ can reduce the value abstraction of i to a representation of $\{0; 1\}$, the only possible integer offset for the array t . At line 5, the abstract semantics $F_{\text{addr}}^{\#}$ can reduce the value abstraction of p to a representation of $\{\&x\}$, as NULL is never a valid pointer value. In both cases, the location abstraction of the address represents only the valid memory locations for the dereference.

5.3 THE CVALUE IMPLEMENTATION

EVA provides abstractions of integer, floating-point and pointer values called *cvalue*. These abstractions are not new: they are inherited from **VALUE**, the former abstract interpreter of FRAMA-C. They handle the whole semantics of expressions, and always produce closed maps of alarms to report undesirable behaviors. As the default value abstractions, they are the privileged means of communication between abstract domains in our analyzer. They are able to express precise alias information between variables, which is very convenient for purely numerical domains.

This section focuses especially on *garbled mixes*, an abstraction designed to soundly approximate the result of any arithmetic operations on pointer values. While these abstractions are not new, their formal semantics and their backward propagators are a contribution of this thesis. This section first describes formally the representation of the cvalue abstractions. It then gives an overview of their semantics, focusing on the addition between pointers and integers. It finally addresses the backward propagators for the same operation.

5.3.1 Basic Representation of Constant Values

As explained in Section 4.3.3, the concrete values of our semantics are functions from memory layouts to C values. In practice, most of the values manipulated by a program are independent from the memory layout —and are thus constant functions in our semantics. Other concrete values only arise when performing integer arithmetic on variable addresses. The cvalue module provides specific and precise abstractions of sets of concrete values that are independent of the memory layout, and another coarser representation for arbitrary concrete values.

The *basic* arithmetic and pointer abstractions stand only for C values independent of the memory layout. They simply describe a set of C values, and their concretization is the set of constant functions, from any memory layouts to a C value in the set they describe. Their representation of sets of C values and their abstract semantics are quite precise.

On the other hand, values depending on the memory layout are captured by an imprecise abstraction called *garbled mix*, whose semantics is postponed to the next subsection.

5.3.1.1 Basic Integer Abstractions

The basic integer abstractions represent sets of integer values. They are a reduced product of sets of discrete integers and integer intervals with congruence.

DISCRETE SETS The sets of integers are not real abstractions: they represent exactly their elements. However, for performance reasons, their cardinal is kept small: above a certain limit, they are automatically converted into integer intervals. Their maximal cardinal is user-configurable², and is 8 by default. These sets are ordered, and are implemented with sorted arrays.

$$\{i_1, i_2, \dots, i_n\} \quad \forall k, i_k \in \mathbb{Z} \quad k < l \Rightarrow i_k < i_l$$

INTEGER INTERVALS These abstract values are themselves a reduced product between intervals and congruence. An integer interval is a pair of two values *min* and *max* in $\mathbb{Z} \cup \{+\infty; -\infty\}$, such that $min \leq max$. A congruence information is encoded with two natural integers *rem* and *mod* of \mathbb{N} , such that $0 < mod$ and $0 \leq rem < mod$. Such an interval and a congruence represent the set of integers between *min* and *max* and congruent to *rem* modulo *mod*. We write this abstraction as:

$$[min..max]rem \% mod$$

A canonical representation of these abstractions is maintained. The lower (respectively upper) bound of the interval is always refined up (resp. down) to the nearest integer validating the congruence. A value carrying no congruence information has $mod = 1$ and $rem = 0$. Any value representing less than 8 integers (by default) is automatically converted into the set of these integers.

CONCRETIZATION These abstractions represent only integer values that do not depend on the memory layout. Formally, their concretizations are the following:

$$\begin{aligned} \gamma_{\text{cval}}(X) &\triangleq \{\lambda_{-}. i \mid i \in \mathbb{Z} \wedge i \in X\} \\ \gamma_{\text{cval}}([min..max]r \% m) &\triangleq \{\lambda_{-}. i \mid i \in \mathbb{Z} \wedge min \leq i \leq max \\ &\quad \wedge i \equiv r \pmod{m}\} \end{aligned}$$

These concretizations go into mathematical numbers. They describe sets of concrete values in any integer type. The set of concrete values of type τ depicted by an abstraction v is then the intersection $(\overline{\mathbb{V}}_{\tau})^{\theta_P} \cap \gamma_{\text{cval}}(v)$.

By a slight abuse of notation, as a basic arithmetic abstraction v describes only constant functions to integer i , we write $i \in v$ for $\lambda_{-}.i \in \gamma_{\text{cval}}(v)$.

CANONIZATION Maintaining a canonical representation of these abstractions requires conversions between discrete sets and intervals. The conversions have to be sound through the concretization. The operator *continuous* expands a set $\{i_1, \dots, i_n\}$ into the smallest interval

² Through the parameter *-val-ilevel*

and congruence abstraction whose concretization contains the constants i_1 to i_n . As the integers of the set are ordered, the interval bounds are i_1 and i_n . The modulo m is the greatest common divisor of the subtractions $i_k - i_1$. Indeed, we have:

$$\exists r, \forall k, i_k \equiv r \pmod{m} \Leftrightarrow \forall k, m \mid (i_k - i_1)$$

And the remainder r can be deduced from the modulo m and any i_k . Conversely, the operator discrete breaks an interval $[i..s]r\%m$ (where i and s are finite) into the set of all integers it describes. As the bounds of the interval satisfy the congruence, both are in the set, as well as all integers $i_1 + k \cdot m$ between i_1 and i_n .

$$\begin{aligned} \text{continuous}(\{i_1, \dots, i_n\}) &= [i_1..i_n]r\%m \\ &\quad \text{with } \begin{cases} m = \gcd(i_2 - i_1, \dots, i_n - i_1) \\ r = i_1 \pmod{m} \end{cases} \\ \text{discrete}([i..s]r\%m) &= \{i, i + m, i + 2m, \dots, s\} \\ &\quad \text{where } i, s \in \mathbb{Z}^2 \end{aligned}$$

Note that the discrete operation is exact, while the continuous can lose much precision. For instance, $\text{continuous}(\{1, 100\}) = [1..100]0\%1$. Finally, the canonize operator maintains a canonical representation of the basic integer abstractions.

$$\begin{aligned} \text{canonize}(\{i_1, \dots, i_n\}) &= \begin{cases} \{i_1, \dots, i_n\} & \text{if } n \leq 8 \\ \text{continuous}(\{i_1, \dots, i_n\}) & \text{otherwise} \end{cases} \\ \text{canonize}([i..s]r\%m) &= \begin{cases} \text{discrete}([i'..s']r\%m) & \text{if } (s' - i')/m < 8 \\ [i'..s']r\%m & \text{otherwise} \end{cases} \\ \text{with } \begin{cases} i' = \begin{cases} \min_j (j \geq i \wedge j \equiv r \pmod{m}) & \text{if } i \in \mathbb{Z} \\ i & \text{otherwise} \end{cases} \\ s' = \begin{cases} \max_t (t \leq s \wedge t \equiv r \pmod{m}) & \text{if } s \in \mathbb{Z} \\ s & \text{otherwise} \end{cases} \end{cases} \end{aligned}$$

5.3.1.2 Basic Floating-point Abstractions

Sets of floating-point values are represented by a floating-point interval, that is a pair of two double-precision floating-point numbers. It represents the set of all floating-point numbers between them, excluding infinite and NaNs. We write them as:

$$[min..max]$$

And the concretization of such an interval is:

$$\gamma_{\text{cval}}([min..max]) \triangleq \{\lambda_. f \mid f \in \mathbb{Q} \wedge min \leq f \leq max\}$$

5.3.1.3 Basic Pointer Abstractions

Concrete pointer values are pairs of a program variable and an integer offset, expressed in bytes. The basic pointer abstractions are maps from variables to basic integer abstractions. The set of variables bound in such a map M is written $\text{dom}(M)$. A map binding the variables x_k to the abstractions o_k is written as follows:

$$\{\{ \&x_k + o_k \}\}_k$$

These maps represent pointer values independent of the memory layout, i.e. pointing to the same memory position in all layouts. Thus, the concretization of a map L contains the set of all constant functions to a pointer value $(\&x, i)$ such that the variable x is in the map, and the integer offset i is in the concretization of $L(x)$.

Furthermore, such a map is also used to depict the integer interpretation of a pointer value. Indeed, the basic arithmetic abstractions are unable to capture the dependence between an integer and a memory location, and cannot represent the integer conversion of a pointer. Instead, we use the maps to represent a set of pointer values, or the integer conversion of these pointer values. Thus, they can represent integer values that are dependent of the memory layout, but whose conversion into a pointer value is independent of the memory layout.

Thus, the maps may denote the possible values of an expression of pointer or integer type. Formally, their concretization is:

$$\begin{aligned} \gamma_{\text{cval}}(\{\{ \&x + o \}\}) &\triangleq \{\lambda_{-}. (\&x, i) \mid \lambda_{-}. i \in \gamma_{\text{cval}}(o)\} \\ &\quad \cup \{\lambda\theta. (\theta(x) + i) \mid \lambda_{-}. i \in \gamma_{\text{cval}}(o)\} \\ \gamma_{\text{cval}}(\{\{ \&x_k + o_k \}\}_{k \in K}) &\triangleq \bigcup_{k \in K} \gamma_{\text{cval}}(\{\{ \&x_k + o_k \}\}) \end{aligned}$$

As for the integer abstractions, this concretization must be intersected with the set of concrete values \mathcal{V} . In particular, some of the pointer values $(\&x_k, i)$ may be invalid and not exist in $\overline{\mathcal{V}}_{\text{ptr}}$. The concretization of a basic pointer abstraction may even contain no pointer values but only integers. For instance, at line 4 of listing 5.2, a sound abstraction of the variable y is the map $\{\{ \&x + [42] \}\}$; its concretization is the singleton $\{\lambda\theta. (\theta(x) + 42)\}$, whose conversion into a pointer value would fail. At line 5, the abstraction of the integer $y - 42$ and of the pointer $(\text{int} *) (y - 42)$ is the same map $\{\{ \&x + [0] \}\}$. It is worth noting that an access through this pointer would be an undefined behavior according to the C standard, but is legitimate in our semantics of `clike`, where we authorize this kind of arithmetic operation on pointer values.

In the implementation, the numerical abstractions are embedded into pointers abstractions as maps from the `NULL` base to the arithmetic abstraction. For the sake of clarity, we keep separate these abstractions in this manuscript.

Listing 5.2: Basic pointer abstraction representing integers

```

1  #include <stdint.h>

   int x = 1;
   uintptr_t y = (uintptr_t)&x + 42;
5  int *z = (int *) (y - 42);

```

5.3.2 Garbled Mix: a Representation of Not Constant Values

The basic pointer abstractions are able to precisely handle the whole semantics of pointers. They also represent the integer conversion of pointers, but are unable to interpret all the integer arithmetic on those concrete values. For instance, the result of an integer multiplication between two variable addresses cannot be represented as a map from a variable to a basic integer representation.

After arithmetic operations that are meaningless on variable addresses, the precise maps for pointer values degenerate into imprecise abstractions in which the offset information is discarded. These new abstractions store only the variable addresses from which the new concrete value has been computed. These abstractions are called *garbled mix*.

5.3.2.1 Definition

Let X be a set of program variables. A garbled mix $\text{GM}(X)$ models all the concrete values whose byte interpretation according to the memory layout depends only on the integer addresses of the variables of X . Such a concrete value has the same byte interpretation in two memory layouts where the variables of X have the same addresses. We consider the byte interpretation of the value to make the symbolic representation of pointer values $(\&x, i)$ dependent of the address of x .

Definition 44. A concrete value V of type τ depends only on the integer addresses of variables in X when for any layouts θ and θ' that coincide on X , the byte interpretations of the C values $V(\theta)$ and $V(\theta')$ are equal.

$$\forall \theta, \theta' \in (\Theta_P)^2, \theta|_X = \theta'|_X \Rightarrow \phi_\tau(V(\theta)) = \phi_\tau(V(\theta'))$$

We denote this property by $V \equiv V|_X$.

Definition 45 (Concretization of garbled mixes). The concretization of a garbled mix is then defined as:

$$\gamma_{\text{cval}}(\text{GM}(X)) = \{V \in \mathcal{V} \mid V \equiv V|_X\}$$

Finally, the widest abstraction is the special garbled mix $\text{GM}(\mathcal{X})$ — \mathcal{X} being the set of the program variables.

Lemma 12. *The concretization of $\text{GM}(\mathcal{X})$ is the set of all concrete values.*

$$\gamma_{\text{cval}}(\text{GM}(\mathcal{X})) = \mathcal{V}$$

Proof. As memory layouts are functions from \mathcal{X} to \mathbb{N} :

$$\begin{aligned} \forall \theta, \theta' \in (\Theta_P)^2, \theta|_{\mathcal{X}} = \theta'|_{\mathcal{X}} &\Rightarrow \theta = \theta' \\ &\Rightarrow \forall V \in \mathcal{V}, \phi_{\tau}(V(\theta)) = \phi_{\tau}(V(\theta')) \end{aligned}$$

And thus $\forall V \in \mathcal{V}, V \equiv V|_{\mathcal{X}} \Rightarrow \forall V \in \mathcal{V}, V \in \gamma_{\text{cval}}(\text{GM}(\mathcal{X}))$. \square

It is worth noting that a garbled mix for an expression expresses nothing on the C values this expression may have, but only specifies the variable addresses on which its values may depend. Thus, a garbled mix may depict the possible values of any expression, regardless of its type.

5.3.2.2 Assumption About Memory Layouts

The semantics of the cvalue abstractions makes an assumption over the possible memory layouts of a program. It requires that the integer address of a variable x cannot be deduced from the address of the other program variables. If this invariant is not guaranteed for a particular program P , then the semantics of the cvalue abstractions is unsound for the analysis of P .

Axiom 1. *Let P be a program, \mathcal{X} the set of variables of this program, and Θ the set of possible memory layouts for these variables. Let x be a variable of \mathcal{X} . The address of x is not determined by the address of all other variables from \mathcal{X} : there is no function $f : \mathbb{N}^{(|\mathcal{X}|-1)} \rightarrow \mathbb{N}$ such that in all layout of Θ , the memory address of x is $f(\theta(\mathcal{X} \setminus \{x\}))$.*

$$\forall x \in \mathcal{X}, \nexists f : \mathbb{N}^{(|\mathcal{X}|-1)} \rightarrow \mathbb{N}, \forall \theta \in \Theta_P, \theta(x) = f(\theta(\mathcal{X} \setminus \{x\}))$$

In other words, for each variable x of the program P , there exists two memory layout θ and θ' such that:

- *the addresses of x in θ and in θ' are different;*
- *but the addresses of all other variables are the same in θ and in θ' .*

$$\forall x \in \mathcal{X}, \exists (\theta, \theta') \in (\Theta_P)^2, \begin{cases} \theta(x) \neq \theta'(x) \\ \forall y \in \mathcal{X}, y \neq x \Rightarrow \theta(y) = \theta'(y) \end{cases}$$

The immediate consequence of this axiom is that a pointer value $(\&x, i)$ belongs to the concretization of a garbled mix $\text{GM}(X)$ if and only if x belongs to X .

Theorem 4. Let X be a set of variables, x be a variable, and i be an integer.

$$\begin{aligned}\lambda_{-}(\&x, i) \in \gamma_{\text{cval}}(\text{GM}(X)) &\Leftrightarrow x \in X \\ \lambda\theta.(\theta(x) + i) \in \gamma_{\text{cval}}(\text{GM}(X)) &\Leftrightarrow x \in X\end{aligned}$$

Proof. Let us remind definition 22 of pointer interpretation:

$$\phi_{\text{ptr}}(\&x, i) = \phi_{\text{address}}(\theta(x) + i) \quad (5.1)$$

We assume first that $x \in X$. Let θ and θ' be two memory layouts such that $\theta|_X = \theta'|_X$. In particular, $\theta(x) = \theta'(x)$.

Let $V = (\lambda_{-}(\&x, i))(\theta)$. Equation 5.1 ensures:

$$\begin{aligned}\phi_{\text{ptr}}(V(\theta)) &= \phi_{\text{address}}(\theta(x) + i) \\ &= \phi_{\text{address}}(\theta'(x) + i) = \phi_{\text{ptr}}(V(\theta'))\end{aligned}$$

Let $V' = \lambda\theta.(\theta(x) + i)$, we also have:

$$\phi_{\text{int}}(V(\theta)) = \phi_{\text{int}}(\theta(x) + i) = \phi_{\text{int}}(\theta'(x) + i) = \phi_{\text{int}}(V(\theta'))$$

In conclusion, V and V' belong to $\gamma_{\text{cval}}(\text{GM}(X))$.

Conversely, let y be a variable such that $\lambda_{-}(\&y, i) \in \gamma_{\text{cval}}(\text{GM}(X))$. Axiom 1 ensures the existence of two memory layouts θ and θ' such that:

$$\begin{cases} \theta(y) \neq \theta'(y) \\ \forall x \in \mathcal{X}, x \neq y \Rightarrow \theta(x) = \theta'(x) \end{cases}$$

And thus $y \notin X \Leftrightarrow \theta|_X = \theta'|_X$.

By definition 45 of garbled mixes, equation 5.1 and the bijectivity of interpretation functions (stated in definition 20):

$$\begin{aligned}\theta|_X = \theta'|_X &\Leftrightarrow \phi_{\text{ptr}}(V(\theta)) = \phi_{\text{ptr}}(V(\theta')) \\ &\Leftrightarrow \phi_{\text{address}}(\theta(y) + i) = \phi_{\text{address}}(\theta'(y) + i) \\ &\Leftrightarrow \theta(y) + i = \theta'(y) + i \\ &\Leftrightarrow \theta(y) = \theta'(y)\end{aligned}$$

Finally, $y \notin X \Leftrightarrow \theta(y) = \theta'(y)$ and $\theta(y) \neq \theta'(y)$ by hypothesis, which implies $y \in X$.

The proof is exactly the same for the concrete value $\lambda\theta.\theta(x) + i$. \square

This proposition allows us to define the lattice structure of cvalue abstractions.

5.3.2.3 Lattice Structure of the Cvalue Abstractions

The diagram of Figure 5.5 defines the lattice structure of the cvalue abstractions. The floating-point intervals are omitted. The top element $\text{GM}(\mathcal{X})$ includes all other abstractions. The inclusion of garbled mixes is the inclusion of their variable sets. All garbled mixes include all basic arithmetic abstractions; a garbled mix includes a pointer map

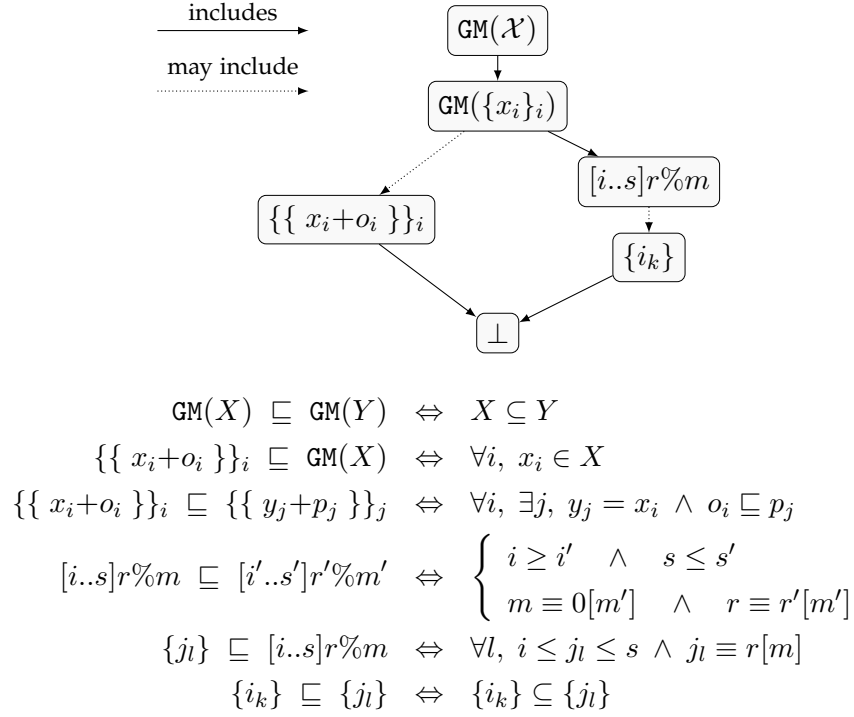


Figure 5.5: Lattice structure of the cvalue abstractions

if all variables of the map belong also to the garbled mix. A map M is more precise than another M' if for all variable, the abstraction of its offset is more precise in M than in M' —considering that the missing variables of a map are bound to \perp . Finally, the lattice structure of the arithmetic cvalue abstractions follows the standard rules of intervals and congruences.

We prove the structure lattice for the operations involving garbled mixes—the other abstractions being more standard.

Lemma 13. *The order relation defined at Figure 5.5 is sound according to the concretization γ_{cval} . For all cvalue abstractions C and C' :*

$$C \sqsubseteq C' \Leftrightarrow \gamma_{\text{cval}}(C) \subseteq \gamma_{\text{cval}}(C')$$

Proof. We prove the case $C' = \text{GM}(X)$, according to the cvalue C .

Case 1. if C is a basic integer or float abstraction:

$$\gamma_{\text{cval}}(C) \subseteq \{\lambda_{\cdot}.r \mid r \in \mathbb{Q}\}$$

Let $V = \lambda_{\cdot}.r$ with $r \in \mathbb{Q}$. The byte interpretation of r does not depend on the memory layout:

$$\forall \theta, \theta' \in (\Theta_P)^2, \phi_{\tau}(V(\theta)) = \phi_{\tau}(r) = \phi_{\tau}(V(\theta'))$$

And by definition 45, for all set X , $V \in \gamma_{\text{cval}}(\text{GM}(X))$.

Thus, for all basic integer or float abstraction C and all set X ,

$$\gamma_{\text{cval}}(C) \subseteq \gamma_{\text{cval}}(\text{GM}(X))$$

Case 2. If C is a map abstraction: $C = \{\{x_i + o_i\}\}_i$.
 Let V be a concrete value in $\gamma_{\text{cval}}(C)$. It has the form $V = \lambda_{\cdot}(\&x_i, j)$ or $V = \lambda\theta.\theta(x_i) + j$.
 By theorem 4, such a concrete value belongs to $\gamma_{\text{cval}}(\text{GM}(X))$ if and only if $x_i \in X$.
 Thus:

$$\gamma_{\text{cval}}(\{\{x_i + o_i\}\}_i) \subseteq \gamma_{\text{cval}}(\text{GM}(X)) \Leftrightarrow \forall i, x_i \in X$$

Case 3. If C is a garbled mix: $C = \text{GM}(Y)$.
 Let V be a concrete value of type τ in $\gamma_{\text{cval}}(\text{GM}(Y))$.

$$\forall \theta, \theta' \in (\Theta_P)^2, \theta|_Y = \theta'|_Y \Rightarrow \phi_{\tau}(V(\theta)) = \phi_{\tau}(V(\theta'))$$

If $Y \subseteq X$, then $\theta|_X = \theta'|_X \Rightarrow \theta|_Y = \theta'|_Y$, and:

$$\forall \theta, \theta' \in (\Theta_P)^2, \theta|_X = \theta'|_X \Rightarrow \phi_{\tau}(V(\theta)) = \phi_{\tau}(V(\theta'))$$

This implies that V also belongs to $\gamma_{\text{cval}}(\text{GM}(X))$.

$$Y \subseteq X \Rightarrow \gamma_{\text{cval}}(\text{GM}(Y)) \subseteq \gamma_{\text{cval}}(\text{GM}(X))$$

Conversely, we assume that $\gamma_{\text{cval}}(\text{GM}(Y)) \subseteq \gamma_{\text{cval}}(\text{GM}(X))$.
 By theorem 4:

$$\begin{aligned} y \in Y &\Rightarrow (\lambda.(\&y, 0)) \in \gamma_{\text{cval}}(\text{GM}(Y)) \\ &\Rightarrow (\lambda.(\&y, 0)) \in \gamma_{\text{cval}}(\text{GM}(X)) \Rightarrow y \in X \end{aligned}$$

And thus $Y \subseteq X$. We have finally:

$$Y \subseteq X \Leftrightarrow \gamma_{\text{cval}}(\text{GM}(Y)) \subseteq \gamma_{\text{cval}}(\text{GM}(X))$$

□

5.3.3 Forward Abstract Semantics of Cvalues

The cvalue semantics of the C operators is divided into the approximation of the corresponding mathematical operation, and the embedding of its result into the type of the concrete operator. In what follows, we omit the emission of alarms and focus on the abstraction of the correct behaviors of the operators. Without detailing the complete cvalue semantics, we use the integer addition as running example.

5.3.3.1 Mathematical Arithmetic

ARITHMETIC ON BASIC ARITHMETIC ABSTRACTIONS The basic arithmetic abstractions represent only constant functions from memory layout to C values. The semantics of an operator \diamond is its pointwise

application on C values under each memory layout independently. As long as the operator on C values does not depend itself on the layouts, its semantics on constant functions lead to a constant function. This is the case for all operators, except for the casts between integer and pointer values.

$$\begin{aligned} \forall i, V_i = \lambda_{-}. v_i \Rightarrow \overline{[\Diamond]}^{\Theta}(V_1, \dots, V_n) &= \lambda\theta. \overline{[\Diamond]}(V_1(\theta), \dots, V_n(\theta)) \\ &= \lambda_{-}. \overline{[\Diamond]}(v_1, \dots, v_n) \end{aligned}$$

Thus, the abstract semantics of basic arithmetic abstractions for these operators produce only basic arithmetic abstractions. The semantics of discrete sets amount to performing the concrete operation on each element of the sets. The semantics of the other abstractions follow the standard arithmetic on intervals and congruences [Moo66; Gra89]. If an operation involves a discrete set and an interval with congruence, the set is treated as an interval abstraction. In any case, the resulting abstraction is revised into its canonical representation.

The abstract semantics for the integer addition is given as example. We use the infix notation $+_{\text{cval}}$ for the cvalue semantics of the mathematical addition.

$$\{x_k\}_{0 \leq k \leq i} +_{\text{cval}} \{y_l\}_{0 \leq l \leq j} = \text{canonize}(\{x_k + y_l\}_{0 \leq k \leq i \wedge 0 \leq l \leq j})$$

$$[i_x..s_x]r_x \% m_x +_{\text{cval}} [i_y..s_y]r_y \% m_y = \text{canonize}([i..s]r \% m)$$

$$\text{where } \begin{cases} i = i_x + i_y & m = \text{gcd}(m_x, m_y) \\ s = s_y + s_y & r = (r_x + r_y) \bmod m \end{cases}$$

$$[i..s]r \% m +_{\text{cval}} \{y_l\}_{0 \leq l \leq j} = [i..s]r \% m +_{\text{cval}} \text{continuous}(\{y_l\}_{0 \leq l \leq j})$$

INTEGER ARITHMETIC ON POINTERS ABSTRACTIONS As variable addresses can be manipulated through integer arithmetic, the pointer abstractions end up describing integer values. As far as possible, maps from variable to integer offsets are used as a precise representation of the integer interpretation of valid pointer values. But for the operations that are meaningless on pointers, the maps degenerate into garbled mix.

We use again the integer addition as an example of the abstract semantics on cvalues. The addition between the integer view of a pointer and an integer is naturally defined as:

$$\overline{[+_{\text{int}}]}^{\Theta}(\lambda\theta. \theta(x) + i, \lambda_{-}. j) = \lambda\theta. \theta(x) + (i + j)$$

This output can be soundly described by a map from the variable x to an abstraction of $(i + j)$. Let M be a basic pointer abstraction and

a be a basic arithmetic abstraction, both representing a set of concrete integer values. We want a map R such that:

$$\begin{aligned} M +_{\text{cval}} a &= R \\ \Rightarrow \forall x \in \gamma_{\text{cval}}(M), \forall y \in \gamma_{\text{cval}}(a), \overline{[+_{\text{int}}]}^{\Theta}(x, y) &\in \gamma_{\text{cval}}(R) \end{aligned}$$

Note that, as the operator $+_{\text{int}}$ is typed, the concrete values x and y must be of integer type. By using the notation $j \in a$ for $\lambda_{-}.j \in \gamma_{\text{cval}}(a)$, this condition can be rewritten as:

$$\forall x \in M, \forall i \in M(x), \forall j \in a, (\lambda\theta. \theta(x) + (i + j)) \in \gamma_{\text{cval}}(R)$$

Thus, the map R binds each variable x in M to a basic integer abstraction of $\lambda_{-}.(i + j)$, for all integers i and j in the abstractions $M(x)$ and a . A precise abstraction for it is $M(x) +_{\text{cval}} a$. The result of the abstract addition is then a precise map:

$$\{\{ \&x_i + o_i \}\}_i +_{\text{cval}} a = \{\{ \&x_i + (o_i +_{\text{cval}} a) \}\}_i$$

However, the map abstractions can only approximate concrete integers that point to the same memory position, regardless of the memory layouts. Yet, most integer operations on variable addresses do not keep this property. For instance, the result of the addition of the address of two variables x and y cannot be represented as a map from a variable to a basic arithmetic abstraction. Indeed, the abstraction bound to x should stand for the address of y , which obviously depends on the memory layout.

$$\overline{[+_{\text{int}}]}^{\Theta}(\lambda\theta. \theta(x), \lambda\theta. \theta(y)) = \lambda\theta. \theta(x) + \theta(y)$$

However, the resulting concrete value $\lambda\theta. \theta(x) + \theta(y)$ belongs to the concretization of the garbled mix $\text{GM}(\{x, y\})$, as a function from the integer address of the variables x and y . More generally, in the abstract semantics of cvalues, the integer addition of two abstractions containing variable addresses degenerates into garbled mix, only retaining the addresses from which the new value has been computed. The complete abstract addition on pointer maps and garbled mix is given by:

$$\begin{aligned} \{\{ \&x_i + o_i \}\}_i +_{\text{cval}} a &= \{\{ \&x_i + (o_i +_{\text{cval}} a) \}\}_i \\ \{\{ \&x_i + o_i \}\}_i +_{\text{cval}} \{\{ \&y_j + p_j \}\}_j &= \text{GM}(\{x_i\}_i \cup \{y_j\}_j) \\ \{\{ \&x_i + o_i \}\}_i +_{\text{cval}} \text{GM}(\{y_j\}_j) &= \text{GM}(\{x_i\}_i \cup \{y_j\}_j) \\ \text{GM}(\{x_i\}_i) +_{\text{cval}} \text{GM}(\{y_j\}_j) &= \text{GM}(\{x_i\}_i \cup \{y_j\}_j) \end{aligned}$$

5.3.3.2 Embedding into the Machine World

The cvalue abstractions describe sets of mathematical values, but the actual operators of our language operate on machine values, encoded

by a fixed and finite number of bytes. Some operations have the same semantics in the mathematical and the machine worlds. This is typically the case for all comparison operators, whose result is always a boolean in $\{0, 1\}$. For the other operators, we need to shrink the abstract cvalue resulting from their mathematical semantics, in order for it to fit inside the operator type. This last step also handles the alarms about integer overflows.

5.3.3.3 Memory Locations of Addresses and Dereference

The cvalue abstractions of the possible memory locations of an address are roughly the same as the basic pointer abstractions of expressions: maps from variables to offset abstractions, except that the offsets are always expressed in bits instead of bytes. This allows the maps to model accurately the location of bitfields. The arithmetic abstractions are ineffective to account for address values, but are used for their integer offsets.

The semantics of the variable maps are the same as before. Composing an address expression and its offset amounts to the addition of a pointer abstraction and an integer abstraction. The abstraction of the address locations is reduced to its valid part. The valid offsets for a variable x , expressed in bit, are $[0..8 \cdot (\text{sizeof}(x) - 1)]0\%8$, except for a bitfield, on which case no assumption can be made on the alignment of the offset. We omit the bitfield case in the following. The reduction of a map amounts to the reduction of all its offsets. The reduction of a garbled mix coming from an expression allows its conversion into a map, from the variables it contains to all possible offsets for them. Indeed, a dereference requires that the pointed location is the same in all memory layouts, which is exactly expressed by such a map.

$$\begin{aligned} \text{offset}(x) &= [0..8 \cdot (\text{sizeof}(x_i) - 1)]0\%8 \\ \text{valid}(\{\{ \&x_i + o_i \}\}_i) &= \{\{ \&x_x + (o_i \sqcap \text{offset}(x)) \}\}_i \\ \text{valid}(\text{GM}(\{x_i\}_i)) &= \{\{ x_i + \text{offset}(x_i) \}\}_i \end{aligned}$$

This forward reduction can be crucial for the precision of the analysis. As we have seen above, a garbled mix tends to propagate to all abstractions, as the composition through any operator of a garbled mix and any other abstraction leads to another garbled mix. However, by requiring the address to be independent of the memory layout, dereferences allow reducing a garbled mix to a precise map representation.

5.3.4 Meet of Garbled Mixes

5.3.4.1 Motivation and General Considerations

Backward propagators on the basic cvalue abstractions can be implemented by following the standard rules of interval, congruence and

pointer arithmetic. The next section is dedicated to an abstract backward semantics involving garbled mixes. As garbled mixes are very loose approximations, their reduction is of particular importance for the analysis precision. However, most reductions on garbled mixes are only possible if the variable addresses are adequately independent from each other in the possible memory layouts of a program. Ideally, we want the guarantee that if a concrete value depends only on the variable addresses in X on one hand, and on the variable addresses in Y in the other, then it depends only on the variable addresses in $X \cap Y$. This would allow us to define the meet between garbled mixes as:

$$\text{GM}(X) \sqcap_{\text{cval}} \text{GM}(Y) = \text{GM}(X \cap Y)$$

This definition is correct if and only if:

$$\gamma_{\text{cval}}(\text{GM}(X)) \cap \gamma_{\text{cval}}(\text{GM}(Y)) \subseteq \gamma_{\text{cval}}(\text{GM}(X \cap Y))$$

If any function from program variables \mathcal{X} to integers \mathbb{N} were a memory layout, this would definitely be the case.

Proof. Let V be a concrete value in both concretizations of $\text{GM}(X)$ and $\text{GM}(Y)$, and let θ and θ' be two functions from \mathcal{X} to \mathbb{N} such that $\theta|_{X \cap Y} = \theta'|_{X \cap Y}$. We define a third function θ'' by:

$$\theta''(x) = \begin{cases} \theta(x) & \text{if } x \in X \\ \theta'(x) & \text{otherwise} \end{cases}$$

As $\theta|_{X \cap Y} = \theta'|_{X \cap Y}$, we have $\theta|_X = \theta''|_X$ and $\theta'|_Y = \theta''|_Y$.

By definition 45, $\phi_\tau(V(\theta)) = \phi_\tau(V(\theta'')) = \phi_\tau(V(\theta'))$.

Thus, for all functions θ and θ' :

$$\theta|_{X \cap Y} = \theta'|_{X \cap Y} \Rightarrow \phi_\tau(V(\theta)) = \phi_\tau(V(\theta'))$$

Which implies that V is in $\gamma_{\text{cval}}(\text{GM}(X \cap Y))$. \square

However, definition 21 imposes some restriction over memory layouts, and the intermediate function used in the proof could not be a memory layout.

5.3.4.2 Difficulties

Variable addresses cannot be completely independent from each others, as definition 21 precludes overlapping between variables. Therefore, if x and y are different variables, then their addresses cannot be the same in a valid memory layout. Axiom 1 excludes the address of a variable to be certainly determined by the addresses of the other variables. This is equivalent to state that the addressing space is strictly larger than the set of program variables. The size of the addressing

space is determined by the number of possible pointer values, namely $2^{\text{sizeof}(\text{ptr})}$, and a variable x takes $\text{sizeof}(x)$ bytes in memory, plus one byte after that must not overlap with an other variable either (see definition 21). Finally, the address 0 is reserved to the NULL pointer.

$$\sum_{x \in \mathcal{X}} (\text{sizeof}(x) + 1) + 1 < 2^{\text{sizeof}(\text{ptr})}$$

However, this is not sufficient for our needs, as illustrated by the following exemple.

Example 10. We consider a set \mathcal{X} of variables, and assume the addressing space to be only one more than $\sum_{x \in \mathcal{X}} (\text{sizeof}(x) + 1) + 1$. Let x be a variable of \mathcal{X} , and θ be any valid memory layout for \mathcal{X} . Then we have:

$$\theta(x) < 3 \Leftrightarrow \forall y \in \mathcal{X} \setminus \{x\}, \theta(y) > 2$$

Thus, the concrete value $V = \lambda\theta.(\theta(x) < 3)$ can be computed in each layout from $\theta(x)$ or from the set of $\theta(y)$ for all other variables y . This means that V belongs to $\text{GM}(\{x\})$ and to $\text{GM}(X \setminus \{x\})$.

Thus, $\text{GM}(\{x\}) \sqcap_{\text{CVal}} \text{GM}(X \setminus \{x\}) \neq \text{GM}(\emptyset)$.

5.3.4.3 New Restrictions on the Addressing Space

As axiom 1 is not strong enough, we reinforce it by axiom 2: it requires that the difference between the size of the addressing space and the size of all program variables is bigger than the size of the biggest program variable.

Axiom 2.

$$\sum_{x \in \mathcal{X}} (\text{sizeof}(x) + 1) + 1 + \max_{x \in \mathcal{X}} (\text{sizeof}(x)) < 2^{\text{sizeof}(\text{ptr})}$$

Lemma 14. Given a program P with a set \mathcal{X} of variables and two memory layouts θ and θ' in Θ_P , axiom 2 ensures the existence of a series of valid memory layouts $\theta_0, \dots, \theta_n$ such that $\theta_0 = \theta$, $\theta_n = \theta'$ and:

$$\forall i \in \{1, \dots, n\}, \exists x \in \mathcal{X}, \forall y \in \mathcal{X}, y \neq x \Rightarrow \theta_i(y) = \theta_{i-1}(y)$$

Proof. The complete proof is given in Annex B. The idea is to shift one variable at a time in θ and θ' until all variables are adjacent, starting at address 1. Then, axiom 2 ensures that it remains enough addressing space to move any variable at the end of the space. The same variable is put at the same address at the end of the addressing space in the two series, and the process can be repeated on the remaining variables (shifting the variables to fill the gap left by the previous move, and choosing a shared variable to move at the released space). \square

Lemma 14 allows us to define the meet of two garbled mixes of disjoint sets of variables.

Lemma 15. *Let X and Y be two sets of program variables. Then:*

$$X \cap Y = \emptyset \Rightarrow \text{GM}(X) \sqcap_{\text{cval}} \text{GM}(Y) = \text{GM}(\emptyset)$$

The concretization of $\text{GM}(\emptyset)$ is the set of concrete values V that are constant according to the layout: $\forall(\theta, \theta'), \phi(V(\theta)) = \phi(V(\theta'))$.

Proof. Let X and Y be two sets of program variables such that $X \cap Y = \emptyset$. Let $V \in \mathcal{V}$ be a concrete value of type τ that belongs to $\gamma_{\text{cval}}(\text{GM}(X))$ and $\gamma_{\text{cval}}(\text{GM}(Y))$. We want to prove that V also belongs to $\gamma_{\text{cval}}(\text{GM}(X \cap Y))$. Let θ and θ' be two memory layouts. By lemma 14, there exists a series of layouts $(\theta_i)_{0 \leq i \leq n}$ from θ to θ' such that:

$$\forall i \in \{1, \dots, n\}, \exists x \in \mathcal{X}, \forall y \in \mathcal{X}, y \neq x \Rightarrow \theta_i(y) = \theta_{i-1}(y)$$

Let $i \in \{1, \dots, n\}$. Let x such that $y \neq x \Rightarrow \theta_i(y) = \theta_{i-1}(y)$.

- Either $x \notin X$, and thus $\theta_{i-1}|_X = \theta_i|_X$. As $V \in \gamma_{\text{cval}}(\text{GM}(X))$, we have $\phi_\tau(V(\theta_i)) = \phi_\tau(V(\theta_{i-1}))$;
- Or $x \in X$, and thus $x \notin Y$. By the same reasoning applied to Y , $\phi_\tau(V(\theta_i)) = \phi_\tau(V(\theta_{i-1}))$.

For each $i \in \{1, \dots, n\}$, $\phi_\tau(V(\theta_i)) = \phi_\tau(V(\theta_{i-1}))$.

Finally, $\phi_\tau(V(\theta)) = \phi_\tau(V(\theta_0)) = \phi_\tau(V(\theta_n)) = \phi_\tau(V(\theta'))$,

and $V \in \gamma_{\text{cval}}(\text{GM}(\emptyset))$. \square

However, axiom 2 is still not sufficient to allow the general definition of garbled mixes. To extend the preceding proof to the general case where $X \cap Y \neq \emptyset$, we need the existence of a series of layouts from θ to θ' such that two successive layouts differ only on one variable x such that $\theta(x) \neq \theta'(x)$. This means that the series of layouts from θ to θ' remains constant on the variables such that $\theta(x) = \theta'(x)$.

Axiom 3.

$$\left(\sum_{x \in \mathcal{X}} (\text{sizeof}(x) + 1) \right) + 1 + (|\mathcal{X}| + 2) \left(\max_{x \in \mathcal{X}} (\text{sizeof}(x)) \right) < 2^{\text{sizeof}(\text{ptr})}$$

Conjecture 1. *Given a set $X \subseteq \mathcal{X}$ of variables and two memory layouts θ and θ' such that $\theta|_X = \theta'|_X$, axiom 3 ensures the existence of a series of valid memory layouts $\theta_0, \dots, \theta_n$ such that $\theta_0 = \theta$, $\theta_n = \theta'$ and:*

$$\forall i \in \{1, \dots, n\}, \exists x \in \mathcal{X} \setminus X, \forall y \in \mathcal{X}, y \neq x \Rightarrow \theta_i(y) = \theta_{i-1}(y)$$

Proof. We only give a general idea of a proof of this conjecture by induction on the number of variables x such that $\theta(x) \neq \theta'(x)$. The result is trivial if there is no such variable. Otherwise, we choose such a variable x , and simply move it in θ by defining $\theta(x) \triangleq \theta'(x)$. If the new function is a layout (i.e. if there is no aliasing conflict), then we are in the inductive case. Otherwise, there is a variable y such that $\theta(y)$ and $\theta'(x)$ are in conflict. In this case, axiom 3 gives us enough addressing space to move y in θ before moving x . \square

Lemma 16. *Let X and Y be two sets of program variables,*

$$\text{GM}(X) \sqcap_{\text{cval}} \text{GM}(Y) = \text{GM}(X \cap Y)$$

Proof. Let $V \in \mathcal{V}$ a concrete value that belongs to $\gamma_{\text{cval}}(\text{GM}(X))$ and $\gamma_{\text{cval}}(\text{GM}(Y))$. Let θ and θ' be two memory layouts such that $\theta|_{X \cap Y} = \theta'|_{X \cap Y}$. By conjecture 1, there exists a series of layouts $(\theta_i)_{0 \leq i \leq n}$ from θ to θ' such that:

$$\forall i \in \{1, \dots, n\}, \exists x \in \mathcal{X} \setminus (X \cap Y), \forall y \in \mathcal{X}, y \neq x \Rightarrow \theta_i(y) = \theta_{i-1}(y)$$

Let $i \in \{1, \dots, n\}$. Let $x \in \mathcal{X} \setminus (X \cap Y)$ such that $y \neq x \Rightarrow \theta_i(y) = \theta_{i-1}(y)$.

- Either $x \notin X$, and $\theta_{i-1}|_X = \theta_i|_X \Rightarrow \phi_\tau(V(\theta_i)) = \phi_\tau(V(\theta_{i-1}))$
- Or $x \notin Y$, and $\theta_{i-1}|_Y = \theta_i|_Y \Rightarrow \phi_\tau(V(\theta_i)) = \phi_\tau(V(\theta_{i-1}))$

For each $i \in \{1, \dots, n\}$, $\phi_\tau(V(\theta_i)) = \phi_\tau(V(\theta_{i-1}))$.

Thus, $\phi_\tau(V(\theta)) = \phi_\tau(V(\theta_0)) = \phi_\tau(V(\theta_n)) = \phi_\tau(V(\theta'))$

and $V \in \gamma_{\text{cval}}(\text{GM}(\emptyset))$. □

5.3.4.4 Discussion on the Address Space Restrictions

In order to define some relevant reductions on garbled mixes, we have made some restrictions on the address space offered by the range of pointer values. We require the address space to be large enough (compared to the space required by the variables of a program), so that no bit of information about the integer addresses of some variables can be inferred from the integer addresses of other variables. Axiom 1 ensures that the integer address of a variable x cannot be computed from the integer addresses of other variables. Axiom 2 ensures that any concrete value that can be computed from the integer addresses of two disjoint sets of variables is a constant functions: it does not depend on the memory layout. In particular, this is the case for the integer address of x , which is not a constant function, and thus cannot be computed from the addresses of other variables. Axiom 3 ensures that a concrete value that can be computed from the integer addresses of two sets X and Y of variables is a function that depends only on the integer addresses of the variables in $X \cap Y$: it has the same C value in two memory layouts in which the variables of $X \cap Y$ have the same addresses. It can thus be computed from these addresses only. It is worth noting that axiom 3 entails axiom 2, which in turn entails axiom 1.

These restrictions on the range of pointer values may seem too strong. However, on modern architectures, the limitation of the address space comes from the hardware rather than from the range of pointer values. When pointers are stored on 64 bits, 16 exabits of storage are required to exhaust the address space. Thus, axiom 3 is

a sensible assumption on 64 bit systems: it holds for any program that does not require several exabits of memory to be executed. In more constrained architectures, our backward propagators on garbled mixes remain sound on programs that do not take advantage of the limited address space to compute the same information about the memory layout from the addresses of different variables. This also seems to be a sensible request.

A last observation can also mitigate the severity of our axioms. In definition 21 of memory layouts, the non-aliasing condition applies simultaneously to all the variables of a program. This choice was made for the sake of simplicity. In practice, variables have a scope, and they do not exist in memory outside their scope. The non-aliasing condition could thus be weakened: at each program point, it must be valid for all the variables in scope at this point. Then, the axioms need to hold only for the set of variables in scope at a program point.

5.3.5 Backward Propagators

We prove here the two lemmas that ensure the soundness of a backward propagation of the garbled mixes involved on an addition. Axiom 2 allows a backward propagation in the specific case of two garbled mixes that do not intersect. Axiom 3 leads to a general backward propagation.

Lemma 17. *Let $X = \{x_i\}$, $Y = \{y_i\}$ and $Z = \{z_i\}$ three sets of variables such that $X \cap Y = \emptyset$. Let M be a basic pointer map such that $\text{dom}(M) = Z$. Assuming axiom 2, if*

$$\begin{array}{ccccc} x & + & y & = & z \\ \cap & & \cap & & \cap \\ \text{GM}(X) & & \text{GM}(Y) & & M \end{array}$$

then

$$\begin{aligned} x &\in \{\{x_i + \top \mid x_i \in Z\}\} \cup \{\{\text{NULL} + \top\}\} \\ y &\in \{\{y_i + \top \mid y_i \in Z\}\} \cup \{\{\text{NULL} + \top\}\} \end{aligned}$$

Proof. Let V_X , V_Y and V_Z be three concrete values of arithmetic type τ such that $V_X + V_Y = V_Z$ and

$$V_X \in \gamma_{\text{cval}}(\text{GM}(X)) \quad V_Y \in \gamma_{\text{cval}}(\text{GM}(Y)) \quad V_Z \in \gamma_{\text{cval}}(M)$$

There exists $z \in Z$ and $i \in \mathbb{N}$ such that $V_Z = \lambda\theta.\theta(z) + i$.

We assume $z \notin Y$ (otherwise, $z \notin X$ as $X \cap Y = \emptyset$, and the proof is symmetric). Let θ and θ' two memory layouts. By lemma 14, there exists a series of layouts $\theta_0, \dots, \theta_n$ such that $\theta_0 = \theta$, $\theta_n = \theta'$ and

$$\forall i \in \{1, \dots, n\}, \exists x_i \in \mathcal{X}, \forall y \in \mathcal{X}, y \neq x_i \Rightarrow \theta_i(y) = \theta_{i-1}(y)$$

Let $i \in \{1, \dots, n\}$.

- If $x_i \notin X$, then $V_X(\theta_i) = V_X(\theta_{i-1})$.
- Otherwise, $x \notin Y$ and $V_Y(\theta_i) = V_Y(\theta_{i-1})$.
 - If $x \neq z$, $V_Z(\theta_i) = V_Z(\theta_{i-1})$. As $V_X = V_Z - V_Y$, we also have $V_X(\theta_i) = V_X(\theta_{i-1})$.
 - Otherwise, $V_X(\theta_i) - V_X(\theta_{i-1}) = \theta_i(z) - \theta_{i-1}(z)$.

In all cases, $V_X(\theta_i) - V_X(\theta_{i-1}) = \theta_i(z) - \theta_{i-1}(z)$.

Thus, $V_X(\theta') - V_X(\theta) = \theta'(z) - \theta(z)$.

This equation holds for any layouts θ and θ' . We can conclude that it exists $k \in \mathbb{N}$ such that for any layout θ :

$$\begin{cases} V_X(\theta) = \theta(z) + k \\ V_Y(\theta) = i - k \end{cases}$$

And thus $V_X \in \gamma_{\text{cval}}(\{\{ z + \top \}\})$ and $V_Y \in \gamma_{\text{cval}}(\{\{ \text{NULL} + \top \}\})$. This is true for any $z \in X \cap Z$, and symmetrically for any $z \in Y \cap Z$. In the general case, we have:

$$\begin{cases} V_X \in \gamma_{\text{cval}}(\{\{ x_i + \top \mid x_i \in Z \}\} \cup \{\{ \text{NULL} + \top \}\}) \\ V_Y \in \gamma_{\text{cval}}(\{\{ y_i + \top \mid y_i \in Z \}\} \cup \{\{ \text{NULL} + \top \}\}) \end{cases}$$

□

As the addition is inversible, the general backward propagator can simply be defined using the forward semantics and the meet operator.

Lemma 18. Let $X = \{x_i\}$, $Y = \{y_i\}$ and $Z = \{z_i\}$ three sets of variables. Let M_X , M_Y and M_Z be three basic pointer maps such that $\text{dom}(M_X) = X$, $\text{dom}(M_Y) = Y$ and $\text{dom}(M_Z) = Z$. Assuming axiom 3, if

$$\begin{array}{ccc} x & + & y & = & z \\ \cap & & \cap & & \cap \\ \text{GM}(X) & & \text{GM}(Y) & & \text{GM}(Z) \\ \text{or } M_X & & \text{or } M_Y & & \text{or } M_Z \end{array}$$

then

$$\begin{aligned} x &\in \text{GM}(X \cap (Y \cup Z)) \\ y &\in \text{GM}(Y \cap (X \cup Z)) \end{aligned}$$

Proof. Let V_X , V_Y and V_Z be three concrete values of arithmetic type τ such that $V_X + V_Y = V_Z$ and

$$V_X \in \gamma_{\text{cval}}(\text{GM}(X)) \quad V_Y \in \gamma_{\text{cval}}(\text{GM}(Y)) \quad V_Z \in \gamma_{\text{cval}}(\text{GM}(Z))$$

Then $V_X = V_Z - V_Y$ and $V_Z - V_Y \in \gamma_{\text{cval}}(Y \cup Z)$.

Thus:

$$V_X \in \gamma_{\text{cval}}(\text{GM}(X)) \cap \gamma_{\text{cval}}(\text{GM}(Y \cup Z)) \subseteq \gamma_{\text{cval}}(\text{GM}(X) \sqcap_{\text{cval}} \text{GM}(Y \cup Z))$$

and by lemma 16:

$$\gamma_{\text{cval}}(\text{GM}(X) \sqcap_{\text{cval}} \text{GM}(Y \cup Z)) \subseteq \gamma_{\text{cval}}(\text{GM}(X \cap (Y \cup Z)))$$

Finally, $V_X \in \gamma_{\text{cval}}(\text{GM}(X \cap (Y \cup Z)))$. The proof is identical for V_Y .

As $M_X \sqsubseteq_{\text{cval}} \text{GM}(X)$:

$$V \in \gamma_{\text{cval}}(M_X) \Rightarrow V \in \gamma_{\text{cval}}(\text{GM}(X))$$

Likewise for M_Y and M_Z , and the proof holds for $V_X \in \gamma_{\text{cval}}(M_X)$, or $V_Y \in \gamma_{\text{cval}}(M_Y)$, or $V_Z \in \gamma_{\text{cval}}(M_Z)$. Naturally, if $V_X \in \gamma_{\text{cval}}(M_X)$ and $V_Y \in \gamma_{\text{cval}}(M_Y)$, there is a more precise backward propagation using pointer arithmetic. \square

CONCLUSION

An abstract semantics of expression is implemented through alarm maps and value abstractions. An alarm map reports the undesirable behaviors that may arise when the evaluation of an expression reaches an illegal operation. A value abstractions represents the valid results of the evaluation of an expression. Forward and backward functions over-approximate the semantics of each operator \diamond of the programming language, linking abstractions of the operands and abstractions of the result. In a language with pointers, the value abstractions can also represent variable addresses and pointer values. On a language where the addresses of variables can be handled as standard integers, the abstract semantics of values must also be able to soundly interpret the integer arithmetic on pointer values. To remain scalable, EVA uses coarse abstractions, named garbled mixes, to represent the integer computations on addresses that cannot be easily represented as standard pointer values. Backward propagators are then important to regain precision on garbled mixes, but they require further assumptions about the address space provided by the range of possible pointer values.

The next chapter describes how a generic engine to compute sound abstractions of complete expressions can be derived from the abstract semantics of operators.

EVALUATION OF EXPRESSIONS

A value abstraction implements a full abstract semantics for all the operators of our language. These semantics issue alarms to report undesirable behaviors; otherwise, they over-approximate their effects by producing abstractions for the result, or for the operands. Likewise, a location abstraction represents the meaning of addresses, by modeling the offset of fields in structures and cells of arrays.

Thus, value and location abstractions allow the evaluation of all expression fragments, except for dereferences. Indeed, dereferences depend on the concrete state in which they occur, more precisely on the content of memories at the location pointed out by the address. This chapter assumes given an abstract semantics for the dereference: a forward and a backward function that link an (abstract) location to the (abstract) value at this location.

$$\begin{aligned} F_{*_{\tau}}^{\#}(S^{\#}) &: \mathbb{L}^{\#} \rightarrow \mathbb{V}^{\#} \\ B_{*_{\tau}}^{\#}(S^{\#}) &: \mathbb{V}^{\#} \rightarrow \mathbb{L}^{\#} \end{aligned}$$

This is actually the main feature required from a state abstraction—also called abstract domain. Unlike the semantics of other operators, these functions depend on the current abstract state $S^{\#}$, representing a set of concrete states. Equipped with such functions, a value abstraction and a location abstraction are sufficient to evaluate any expression, by successive applications of the appropriate abstract semantics.

This chapter is dedicated to the inner workings of the [EVA](#) evaluator and the strategies it uses, regardless of the underlying abstractions it relies on. We especially focus on the trade-off reached for the interweaving of forward and backward evaluations. We also mention the partitioning of evaluations, to mitigate the need for relational information.

6.1 EVALUATION AND VALUATION

This section presents the generic evaluator of [EVA](#), that can be instantiated above any value and state abstractions. The previous chapter was dedicated to the value abstractions of [EVA](#). This section first explains the requirements that the state abstraction must fulfill, and then presents the operating principles of the evaluator, that acts by atomic update of maps from expressions to value abstractions, named *valuations*.

```

1 module Make
  (Value : Abstract_value.S)
  (Loc : Abstract_location.S with type value = Value.t)
  (Domain : Abstract_domain.S with type value = Value.t
5                                and type location = Loc.location)

```

Figure 6.1: The evaluation functor

6.1.1 A Generic Functor

In *EVA*, the generic evaluator is an OCaml functor from a module of value abstractions, a module of location abstractions and a module of state abstractions. These abstractions have to be compatible with each other. The location semantics use the value abstractions of the expressions appearing in addresses. Conversely, on C expressions such as `addrof(a)` and `startof(a)`, the abstract location of the address a must be converted into an abstract value for the expression. Finally, the abstract semantics of dereferences, provided by the state abstraction, are functions mapping a location to a value abstraction.

6.1.2 Abstract Domain Requirement

We present here the features required from an abstract domain (our state abstractions) in order to enable the evaluation of expressions. Chapter 7 is fully dedicated to abstract domains, and expounds their formalization with more details and examples. An abstract domain \mathbb{D} is a collection of state abstractions: each abstract state represents a set of concrete states. As usual, we assume the existence of a concretization function $\gamma_{\mathbb{D}}$ that links abstract states to concrete states.

$$\gamma_{\mathbb{D}} : \mathbb{D} \rightarrow \mathcal{P}(\mathcal{S})$$

The evaluation of an expression depends on an abstract state, describing the possible concrete states in which the evaluation occurs. The domain implements different queries that supplies the evaluator with value abstractions.

6.1.2.1 Abstract Semantics of Dereferences

The first query of an abstract domain is the abstract semantics of dereference, mandatory to complete the evaluation of expressions. This semantics takes place in an abstract state D . The forward semantics $F_{*r}^{\#}(D)$ receives an abstraction of the possible memory locations of the lvalue being dereferenced; it computes a sound value abstraction of the concrete values that may be stored in these locations in all the states represented by the abstract state D . $F_{*r}^{\#}(D)$ also produces the alarms that ensure the validity of the location, and that the con-

tents of the read memory slice are proper (i.e. not indeterminate in C parlance, and in particular initialized).

Thus, in an abstract state D , from an abstract location l , the value abstraction and the alarm map produced by $F_{*_{\tau}}^{\#}(D)$ must be a sound approximation of $\llbracket *_{\tau} c \rrbracket^{\Theta}(S)$, for any concrete state S in $\gamma_{\mathbb{D}}(D)$ and any constant c in $\gamma_{\mathbb{V}}(l)$.

$$\begin{aligned} F_{*_{\tau}}^{\#} : \mathbb{D} \rightarrow \mathbb{L}^{\#} \rightarrow \mathbb{V}^{\#} \times \mathbb{A} \\ \forall c \in \gamma_{\mathbb{L}}(l), \forall S \in \gamma_{\mathbb{D}}(D), F_{*_{\tau}}^{\#}(D, l) \models_{\mathbb{V} \times \mathbb{A}} \llbracket *_{\tau} c \rrbracket^{\Theta}(S) \end{aligned}$$

See definition 40 for the formalization of $\models_{\mathbb{V} \times \mathbb{A}}$. It is worth noting that this forward semantics only abstracts the dereference of constant locations c in $\gamma_{\mathbb{L}}(l)$, which must be valid locations to avoid undesirable behaviors. It does not handle the evaluation of an address into a memory location, whose abstract semantics is provided by location abstractions.

The backward counterpart $B_{*_{\tau}}^{\#}(D, l, v)$ of this abstract semantics tries to reduce the abstract location l , knowing a value abstraction v of its dereference. The concretization of the new abstract location must contain all the concrete locations that lead to a concrete value in $\gamma_{\mathbb{V}}(v)$ in the concrete states in $\gamma_{\mathbb{D}}(D)$.

$$\begin{aligned} B_{*_{\tau}}^{\#} : \mathbb{D} \rightarrow \mathbb{L}^{\#} \rightarrow \mathbb{V}^{\#} \rightarrow \mathbb{L}^{\#} \\ \gamma_{\mathbb{L}}(B_{*_{\tau}}^{\#}(D, l, v)) \supseteq \{c \in \gamma_{\mathbb{L}}(l) \mid \exists S \in \gamma_{\mathbb{D}}(D), \llbracket *_{\tau} c \rrbracket^{\Theta}(S) \in \gamma_{\mathbb{V}}(v)\} \end{aligned}$$

The forward and backward semantics of dereferences provided by an abstract domain are respectively illustrated in Sections 7.1.1 and 7.2.1.

6.1.2.2 Additional Queries

As abstraction of the concrete states, the abstract domain is naturally able to provide an abstract semantics for dereference. However, an abstract domain may infer some relevant properties on other expressions than dereferences. Thus, an abstract domain also supplies the evaluator with alarm maps and value abstractions for arbitrary expressions. This is the query $F_{\mathbb{D}}^{\#}(D, e)$, which computes sound abstractions of the concrete evaluation of an expression e in an abstract state D

$$\begin{aligned} F_{\mathbb{D}}^{\#} : \mathbb{D} \rightarrow \text{expr} \rightarrow \mathbb{V}^{\#} \times \mathbb{A} \\ \forall S \in \gamma_{\mathbb{D}}(D), F_{\mathbb{D}}^{\#}(D, e) \models_{\mathbb{V} \times \mathbb{A}} \llbracket e \rrbracket^{\Theta}(S) \end{aligned}$$

This additional query is detailed in Section 7.1.2.

6.1.3 Valuations

The generic evaluator manipulates partial maps from expressions to value abstractions. Such a map stores the results of all computations done during an evaluation, including all intermediate steps. They make available the value abstraction computed for each expression fragment processed by the evaluator. These maps are called valuations, ranged over by \mathcal{E} . They are abstractions of sets of concrete states.

6.1.3.1 Formalization

Definition 46 (Valuations). A valuation is a partial function from expressions to value abstractions. A valuation \mathcal{E} represents all the concrete states S for which the abstraction $\mathcal{E}(e)$ is a sound approximation of the concrete evaluation $\llbracket e \rrbracket^\Theta(S)$, for all expressions e in the valuation.

$$\gamma_{\mathbb{E}}(\mathcal{E}) \triangleq \{S \mid \forall e \in \text{dom}(\mathcal{E}), \llbracket e \rrbracket^\Theta(S) \in \gamma_v(\mathcal{E}(e))\}$$

Notation 6. The set of valuations is denoted \mathbb{E} .

$$\mathbb{E} = \text{expr} \rightarrow \mathbb{V}^\#$$

By extension, we consider that any missing expression in a valuation is implicitly bound to the special top abstraction $\top_v + \Omega$, whose concretization is the set of concrete values plus the error value. Then, a valuation \mathcal{E} abstracts the set of concrete states S such that for *all* expression e , $\llbracket e \rrbracket^\Theta(S) \in \gamma_v(\mathcal{E}(e))$. However, notice that a valuation only contains explicitly value abstraction for some expressions, without any error abstraction. Thus, a valuation only abstracts the states in which the concrete evaluation of these expressions succeeds. To exclude the potential error cases—and the states leading to them—the evaluator also produces alarms, separately from valuations.

In order to store evenly all computations of an evaluation, a valuation is actually a pair of a two maps, one from expressions to value abstractions and the other from addresses to locations abstractions. Both maps work in the same way, so we omit the second map in the formalization for the sake of simplicity. Also, the two maps are used to record some other information provided or used by the evaluator. However, these additional data impact neither the concretization nor the soundness of the valuation, and may safely be omitted from their formalization.

6.1.3.2 Lattice Structure

The valuations naturally get a lattice structure. A valuation \mathcal{E}_1 is more precise than another valuation \mathcal{E}_2 whenever \mathcal{E}_1 contains at least all expressions constrained by \mathcal{E}_2 , and that their abstractions in \mathcal{E}_1

are more precise than those in \mathcal{E}_2 . The top valuation is the empty valuation, that constrains the value of no expression. The join and the meet are defined by the pointwise application of the corresponding operations in the underlying abstractions. Finally, would a valuation bind an expression to the bottom abstraction, then the valuation is itself \perp .

Definition 47 (Lattice of valuations). The lattice of valuations is defined as:

$$\begin{aligned}
\mathcal{E} \sqsubseteq_{\mathbb{E}} \mathcal{E}' &\Leftrightarrow \forall e \in \text{dom}(\mathcal{E}'), e \in \text{dom}(\mathcal{E}) \wedge \mathcal{E}(e) \sqsubseteq_{\mathbb{V}} \mathcal{E}'(e) \\
\text{dom}(\mathcal{E} \sqcup_{\mathbb{E}} \mathcal{E}') &\triangleq \text{dom}(\mathcal{E}) \cup \text{dom}(\mathcal{E}') \\
\text{dom}(\mathcal{E} \sqcap_{\mathbb{E}} \mathcal{E}') &\triangleq \text{dom}(\mathcal{E}) \cap \text{dom}(\mathcal{E}') \\
\mathcal{E} \sqcup_{\mathbb{E}} \mathcal{E}' &\triangleq \lambda e. \mathcal{E}(e) \sqcup_{\mathbb{V}} \mathcal{E}'(e) \quad \text{if } e \in \text{dom}(\mathcal{E} \sqcup_{\mathbb{E}} \mathcal{E}') \\
\mathcal{E} \sqcap_{\mathbb{E}} \mathcal{E}' &\triangleq \lambda e. \begin{cases} \mathcal{E}(e) \sqcap_{\mathbb{V}} \mathcal{E}'(e) & \text{if } e \in \text{dom}(\mathcal{E}) \cap \text{dom}(\mathcal{E}') \\ \mathcal{E}(e) & \text{if } e \in \text{dom}(\mathcal{E}) \setminus \text{dom}(\mathcal{E}') \\ \mathcal{E}'(e) & \text{if } e \in \text{dom}(\mathcal{E}') \setminus \text{dom}(\mathcal{E}) \end{cases}
\end{aligned}$$

Lemma 19. *The valuation order is consistent with the concretization:*

$$\mathcal{E} \sqsubseteq_{\mathbb{E}} \mathcal{E}' \Rightarrow \gamma_{\mathbb{E}}(\mathcal{E}) \subseteq \gamma_{\mathbb{E}}(\mathcal{E}')$$

Proof. This lemma relies on the same soundness property of the underlying value lattice: $v \sqsubseteq_{\mathbb{V}} v' \Rightarrow \gamma_{\mathbb{V}}(v) \subseteq \gamma_{\mathbb{V}}(v')$. Thus, we can rewrite:

$$\mathcal{E} \sqsubseteq_{\mathbb{E}} \mathcal{E}' \Rightarrow \text{dom}(\mathcal{E}') \subseteq \text{dom}(\mathcal{E}) \wedge \forall e \in \text{dom}(\mathcal{E}'), \gamma_{\mathbb{V}}(\mathcal{E}(e)) \subseteq \gamma_{\mathbb{V}}(\mathcal{E}'(e))$$

We assume $\mathcal{E} \sqsubseteq_{\mathbb{E}} \mathcal{E}'$. Then, for any concrete state $S \in \mathcal{S}$, we have:

$$\begin{aligned}
\forall e \in \text{dom}(\mathcal{E}), \overline{\llbracket e \rrbracket}^{\Theta}(S) &\in \gamma_{\mathbb{V}}(\mathcal{E}(e)) \\
\Rightarrow \forall e \in \text{dom}(\mathcal{E}'), \overline{\llbracket e \rrbracket}^{\Theta}(S) &\in \gamma_{\mathbb{V}}(\mathcal{E}(e)) \\
\Rightarrow \forall e \in \text{dom}(\mathcal{E}'), \overline{\llbracket e \rrbracket}^{\Theta}(S) &\in \gamma_{\mathbb{V}}(\mathcal{E}'(e))
\end{aligned}$$

Thus, by definition 46, $\gamma_{\mathbb{E}}(\mathcal{E}) \subseteq \gamma_{\mathbb{E}}(\mathcal{E}')$. □

Lemma 20. *The valuation join is an over-approximation of the union of concrete states.*

$$\gamma_{\mathbb{E}}(\mathcal{E}) \cup \gamma_{\mathbb{E}}(\mathcal{E}') \subseteq \gamma_{\mathbb{E}}(\mathcal{E} \sqcup_{\mathbb{E}} \mathcal{E}')$$

Proof. Let $\mathcal{E}'' = \mathcal{E} \sqcup_{\mathbb{E}} \mathcal{E}'$, and let $S \in \gamma_{\mathbb{E}}(\mathcal{E}) \cup \gamma_{\mathbb{E}}(\mathcal{E}')$. We need to prove that $S \in \gamma_{\mathbb{E}}(\mathcal{E}'')$.

Let $e \in \text{dom}(\mathcal{E}'')$. By definition of the join, $e \in \text{dom}(\mathcal{E}) \cap \text{dom}(\mathcal{E}')$. By definition of the concretization of valuations:

$$\overline{\llbracket e \rrbracket}^{\Theta}(S) \in \gamma_{\mathbb{V}}(\mathcal{E}(e)) \quad \vee \quad \overline{\llbracket e \rrbracket}^{\Theta}(S) \in \gamma_{\mathbb{V}}(\mathcal{E}'(e))$$

By soundness property of the join of value abstractions:

$$\llbracket e \rrbracket^\Theta(S) \in \gamma_v(\mathcal{E}(e) \sqcup_v \mathcal{E}'(e))$$

As $\mathcal{E}''(e) = \mathcal{E}(e) \sqcup_v \mathcal{E}'(e)$, we finally have:

$$\forall e \in \text{dom}(\mathcal{E}''), \llbracket e \rrbracket^\Theta(S) \in \gamma_v(\mathcal{E}''(e))$$

Which implies $S \in \gamma_{\mathbb{E}}(\mathcal{E}'')$. \square

Lemma 21. *The valuation meet is an over-approximation of the intersection of concrete states.*

$$\gamma_{\mathbb{E}}(\mathcal{E}) \cap \gamma_{\mathbb{E}}(\mathcal{E}') \subseteq \gamma_{\mathbb{E}}(\mathcal{E} \sqcap_{\mathbb{E}} \mathcal{E}')$$

Proof. Let $\mathcal{E}'' = \mathcal{E} \sqcap_{\mathbb{E}} \mathcal{E}'$, and let $S \in \gamma_{\mathbb{E}}(\mathcal{E}) \cap \gamma_{\mathbb{E}}(\mathcal{E}')$. We need to prove that $S \in \gamma_{\mathbb{E}}(\mathcal{E}'')$.

Let $e \in_{\mathbb{D}} \text{dom}(\mathcal{E}'')$. By definition of the meet, $e \in \text{dom}(\mathcal{E}) \cup \text{dom}(\mathcal{E}')$.

Case 1. $e \in \text{dom}(\mathcal{E}) \cap \text{dom}(\mathcal{E}')$

By definition of the concretization,

$$\llbracket e \rrbracket^\Theta(S) \in \gamma_v(\mathcal{E}(e)) \quad \wedge \quad \llbracket e \rrbracket^\Theta(S) \in \gamma_v(\mathcal{E}'(e))$$

By soundness property of the meet of value abstractions:

$$\llbracket e \rrbracket^\Theta(S) \in \gamma_v(\mathcal{E}(e) \sqcap_v \mathcal{E}'(e)) = \gamma_{\mathbb{E}}(\mathcal{E}'')$$

Case 2. $e \in \text{dom}(\mathcal{E}) \setminus \text{dom}(\mathcal{E}')$

By definition of the concretization,

$$\llbracket e \rrbracket^\Theta(S) \in \gamma_v(\mathcal{E}(e)) = \gamma_{\mathbb{E}}(\mathcal{E}'')$$

Case 3. $e \in \text{dom}(\mathcal{E}') \setminus \text{dom}(\mathcal{E})$: symmetric case.

In all three cases, $\forall e \in \text{dom}(\mathcal{E}''), \llbracket e \rrbracket^\Theta(S) \in \gamma_{\mathbb{E}}(\mathcal{E}'')$.

Thus, $S \in \gamma_{\mathbb{E}}(\mathcal{E}'')$. \square

6.1.4 Forward and Backward Evaluations

6.1.4.1 Definitions

A forward evaluation of an expression is an abstract transposition of its concrete evaluation, through the complete semantics induced by the value, location and state abstractions. It is a bottom-up propagation of value (and location) abstractions, from the leaves (the constants) to the root of the expression. The constants are first converted into value abstractions, including the integer addresses of the variables appearing in the expression. Then, the forward evaluation relies on the abstract forward semantics of operators and dereference

to approximate the value of each subterm, until it reaches the root expression. It also accumulates the alarms issued by these semantics at each step. The alarms ensuring the absence of undesirable behavior during the concrete evaluation of the expression are the union of all those alarms.

Conversely, a backward evaluation is a top-down propagation that aims at reducing the value (and location) abstractions computed for the subterms of an expression, knowing an abstraction of the value of the expression. It relies on the abstract backward semantics for operators and dereference, which learn information from the result of an operation – possibly enabling reductions on the arguments.

Both evaluations operate on a valuation. At the beginning, the valuation is empty by default, but an evaluation can also be initiated with a valuation that already constrains the value of some expressions. At each step, the forward or the backward evaluation fills this valuation with the computed abstractions. At the end, the valuation is the main result of the evaluation: it contains all the computed abstractions, including one for the root expression. Thus, forward and backward evaluations are functions from an initial valuation to an updated valuation. In addition, the forward evaluation produces a map of alarms, and the backward evaluation takes as argument a value abstraction for the root expression.

Notation 7 (Forward and backward evaluation). For an abstract state D , we write respectively $\llbracket e \rrbracket_D^\#$ and $\llbracket e \rrbracket_D^\leftarrow^\#$ the forward and the backward evaluation of the expression e .

$$\begin{aligned} \llbracket \cdot \rrbracket_D^\# &: \mathbb{E} \rightarrow \mathbb{E} \times \mathbb{A} \\ \llbracket \cdot \rrbracket_D^\leftarrow^\# &: \mathbb{E} \times \mathbb{V}^\# \rightarrow \mathbb{E} \end{aligned}$$

Forward and backward evaluations over-approximate the concrete evaluation of e in all the concrete states in the concretization of D . Their soundness, defined below, relies on the concretization of the abstract state D .

6.1.4.2 Soundness and Precision Requirements

Basically, both abstract evaluations must preserve the soundness of the valuation. An evaluation plays out in an abstract state, and starts with an initial valuation. Both the abstract state and the valuation are abstractions of a set of concrete states. An abstract evaluation approximates the possible concrete value of expressions in the concrete states represented by both abstractions.

Notation 8. Let D be an abstract state and \mathcal{E} be a valuation. We denote by $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$ the intersection of the concretization of an abstract state A and the concretization of a valuation \mathcal{E} .

$$\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}) \triangleq \gamma_{\mathbb{D}}(D) \cap \gamma_{\mathbb{E}}(\mathcal{E})$$

Proposition 1 (Soundness of a forward evaluation). *A sound forward evaluation $\llbracket e \rrbracket_D^\#(\mathcal{E})$ produces an alarm map \mathbf{A} and a valuation \mathcal{E}' such that, for any state S in $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$:*

- \mathbf{A} is a sound abstraction of the undesirable behaviors of $\llbracket e \rrbracket^\Theta(S)$;
- if the expression e evaluates without error in S , the valuation \mathcal{E}' is still a sound abstraction of S . Otherwise, this state is excluded by some assertions of the alarm map.

$$\llbracket e \rrbracket_D^\#(\mathcal{E}) = \mathcal{E}', \mathbf{A} \Rightarrow \forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \begin{cases} \mathbf{A} \models_{\mathbf{A}} \llbracket e \rrbracket^\Theta(S) \\ \llbracket e \rrbracket^\Theta(S) \neq \Omega \Rightarrow S \in \gamma_{\mathbb{E}}(\mathcal{E}') \end{cases}$$

A backward evaluation ignores the undesirable behaviors and deals only with value (and location) abstractions. It takes a value abstraction v for the expression e , and updates a valuation accordingly.

Proposition 2 (Soundness of a backward evaluation). *A sound backward evaluation $\llbracket \overleftarrow{e} \rrbracket_D^\#(\mathcal{E}, v)$ produces a valuation \mathcal{E}' which is a sound abstraction of all concrete states of $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$ in which e evaluates to a concrete value in $\gamma_v(v)$.*

$$\llbracket \overleftarrow{e} \rrbracket_D^\#(\mathcal{E}, v) = \mathcal{E}' \Rightarrow \forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \llbracket e \rrbracket^\Theta(S) \in \gamma_v(v) \Rightarrow S \in \gamma_{\mathbb{E}}(\mathcal{E}')$$

In both cases, the condition on the produced valuation \mathcal{E}' implies that the abstraction bound in \mathcal{E}' to any expression is a correct approximation of its possible concrete value in some states of $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$. For a forward evaluation, these are the states in which the concrete evaluation succeeds. For a backward evaluation, these are the states for which the given value abstraction represents the resulting value of the concrete evaluation. However, in both cases, there is no guarantee that the new valuation actually contains an abstraction for the requested expression. In fact, according to the soundness requirement, the empty valuation is always sound as the result of a forward or a backward evaluation. Thus, we also impose some precision requirements to an abstract evaluation.

Proposition 3 (Precision of forward and backward evaluations). *A precise forward evaluation $\llbracket e \rrbracket_D^\#(\mathcal{E})$ or backward evaluation $\llbracket \overleftarrow{e} \rrbracket_D^\#(\mathcal{E}, v)$ produces a valuation \mathcal{E}' such that:*

- the updated valuation \mathcal{E}' is more precise than the initial valuation \mathcal{E} ;
- the updated valuation \mathcal{E}' contains an abstraction of the requested expression e ;
- in the case of the backward evaluation $\llbracket \overleftarrow{e} \rrbracket_D^\#(\mathcal{E}, v)$, the value abstraction bound to e in the new valuation \mathcal{E}' is at least as precise as the given value v .

$$\begin{aligned} \llbracket e \rrbracket_D^\#(\mathcal{E}) = \mathcal{E}', \mathbf{A} &\Rightarrow \mathcal{E}' \sqsubseteq_{\mathbb{E}} \mathcal{E} \wedge e \in \text{dom}(\mathcal{E}') \\ \llbracket \overleftarrow{e} \rrbracket_D^\#(\mathcal{E}, v) = \mathcal{E}' &\Rightarrow \mathcal{E}' \sqsubseteq_{\mathbb{E}} \mathcal{E} \wedge e \in \text{dom}(\mathcal{E}') \wedge \mathcal{E}'(e) \sqsubseteq v \end{aligned}$$

In practice, all subterms of the requested expression appear in the valuation produced by our evaluator. However, the abstractions bound to some subterms or to the requested expression itself may be very imprecise, and may even be the top value abstraction.

6.1.5 Atomic Updates of a Valuation

An evaluation is implemented as a series of atomic updates of a valuation, following the abstract semantics applied to the abstractions it already contains. Definition 48 formally defines the atomic forward and backward steps at the level of an expression e , which may be the root expression to be evaluated, or any of its subterms. Let \mathcal{E} be the current valuation and D the current abstract state. If the expression e is an operation $\diamond(e_1, \dots, e_n)$, the forward step binds e to the meet of the value abstractions given by the semantics $F_\diamond^\#$ applied to $\mathcal{E}(e_1)$ to $\mathcal{E}(e_n)$, and by the query $F_{\mathbb{D}}^\#(D, \diamond(\vec{e}_i))$ of the abstract domain. The backward step binds sequentially the expressions e_1 to e_n to the abstractions given by the semantics $B_\diamond^\#$ applied to $\mathcal{E}(e_1)$, \dots , $\mathcal{E}(e_n)$ and $\mathcal{E}(e)$. If one of the expressions is not bound in the map, the top value abstraction \top_v is used instead. The same principles apply to the dereference, with the abstract semantics $F_{*\tau}^\#$ and $B_{*\tau}^\#$ provided by the abstract domain. We omit the computation of addresses, which follows likewise the abstract semantics of location abstractions, with the same requirements as for the value abstractions. Finally, if the expression e is a constant c , the forward step binds e to the abstract embedding $c^\#$ of the constant, while the backward step is the identity.

Also, the update of a valuation is conservative: when a new abstraction v is computed for an expression e at any step, the update operation of valuations meets the new abstraction v with the previous abstraction bound at e in the valuation.

Definition 48 (Atomic update of a valuation). In an abstract state D and the valuation \mathcal{E} , the forward and backward atomic updates of a valuation are defined as:

$$\begin{aligned}
\mathcal{E}(e) &\triangleq \begin{cases} \mathcal{E}(e) & \text{if } e \in \text{dom}(\mathcal{E}) \\ \top_v & \text{otherwise} \end{cases} \\
\text{update}(\mathcal{E}, e, v) &\triangleq \mathcal{E}[e \mapsto \mathcal{E}(e) \sqcap_v v] \\
\mathcal{F}_\diamond^\#(\diamond(\vec{e}_i)) &\triangleq \text{fst}(\mathbf{F}_\diamond^\#(\overrightarrow{\mathcal{E}(e_i)})) \sqcap_v \text{fst}(\mathbf{F}_\mathbb{D}^\#(D, \diamond(\vec{e}_i))) \\
\text{forward}(\mathcal{E}, e) &\triangleq \begin{cases} \text{update}(\mathcal{E}, e, \mathcal{F}_\diamond^\#(\diamond(\vec{e}_i))) & \text{if } e = \diamond(\vec{e}_i) \\ \text{update}(\mathcal{E}, e, \text{fst}(\mathbf{F}_{*_\tau}^\#(\mathcal{E}(a)))) & \text{if } e = *_\tau a \\ \text{update}(\mathcal{E}, e, c^\#) & \text{if } e = c \in \overline{\mathbb{V}} \end{cases} \\
\text{backward}(\mathcal{E}, e) &\triangleq \begin{cases} \text{update}(\mathcal{E}, \vec{e}_i, \mathbf{B}_\diamond^\#(\overrightarrow{\mathcal{E}(e_i)}, \mathcal{E}(e))) & \text{if } e = \diamond(\vec{e}_i) \\ \text{update}(\mathcal{E}, a, \mathbf{B}_{*_\tau}^\#(\mathcal{E}(a), \mathcal{E}(e))) & \text{if } e = *_\tau a \\ \mathcal{E} & \text{if } e = c \in \overline{\mathbb{V}} \end{cases}
\end{aligned}$$

For the backward step, we wrote $\text{update}(\mathcal{E}, \vec{e}, \vec{v})$ for the multiple updates of a vector of expressions \vec{e} to a vector of abstractions \vec{v} in the valuation \mathcal{E} .

Any atomic step preserves the following invariants on the valuation:

- the updated valuation is at least as precise as the previous valuation;
- the updated valuation is an abstraction of the same concrete states as the previous valuation when the concrete evaluation succeeds.

Lemma 22. *In an abstract state D and an initial valuation \mathcal{E} , a forward or backward atomic step only performs updates of the valuation $\text{update}(\mathcal{E}, e, v)$ such that the value v is a sound approximation of the concrete evaluation of the expression e when it does not fail:*

$$\forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \llbracket e \rrbracket^\Theta(S) \in \gamma_v(v) \cup \{\Omega\}$$

Proof. This lemma stems from the soundness of the abstract semantics used to compute the value abstraction v . The proof is similar for each case of the definition of the forward or backward functions. We prove it for the forward evaluation of the application of a value operation. Let $\vec{e}_i = e_1, \dots, e_n$ be n expressions and \diamond an n -ary operator. We need to prove that either the evaluation of $\diamond(\vec{e}_i)$ raises the error value, or:

$$\llbracket \diamond(\vec{e}_i) \rrbracket^\Theta(S) \in \gamma_v(\text{fst}(\mathbf{F}_\diamond^\#(\overrightarrow{\mathcal{E}(e_i)})) \sqcap_v \mathbf{F}_\mathbb{D}^\#(D, \diamond(\vec{e}_i)))$$

From definition 52 of the query $F_{\mathbb{D}}^{\#}$ (see also Section 6.1.2), we have:

$$\overline{\llbracket \diamond(\vec{e}_i) \rrbracket}^{\Theta}(S) \in \gamma_v(F_{\mathbb{D}}^{\#}(D, \diamond(\vec{e}_i))) \cup \{\Omega\} \quad (6.1)$$

And from definition 42 of the semantics of value:

$$\forall \vec{v} \in (\mathbb{V}^{\#})^n, \overline{\llbracket \diamond \rrbracket}^{\Theta}(\gamma_v(\vec{v})) \subseteq \gamma_v(\mathbf{fst}(F_{\diamond}^{\#}(\vec{v}))) \quad (6.2)$$

And from the definition 46 of valuation soundness:

$$\forall e \in \text{dom}(\mathcal{E}), \forall S \in \gamma_{\mathbb{E}}(\mathcal{E}), \overline{\llbracket e \rrbracket}^{\Theta}(S) \in \gamma_v(\mathcal{E}(e)) \quad (6.3)$$

Finally, the definition of concrete evaluations given by Figure 4.8 is:

$$\forall S \in \mathcal{S}, (\forall i, \overline{\llbracket e_i \rrbracket}^{\Theta}(S) = V_i \neq \Omega) \Rightarrow \overline{\llbracket \diamond(\vec{e}_i) \rrbracket}^{\Theta}(S) \triangleq \overline{\llbracket \diamond \rrbracket}^{\Theta}(\vec{V}_i) \quad (6.4)$$

If the evaluation of $\diamond(\vec{e}_i)$ does not fail, then the evaluation of each expression e_i does not fail either. By 6.3 and by defining $\mathcal{E}(e) = \top_v$ for all expressions e outside the domain of \mathcal{E} , we have

$$\forall S \in \gamma_{\mathbb{E}}(\mathcal{E}), \overline{\llbracket e_i \rrbracket}^{\Theta}(S) \in \gamma_v(\mathcal{E}(e_i)) \quad (6.5)$$

And then:

$$\begin{aligned} \forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \overline{\llbracket \diamond(\vec{e}_i) \rrbracket}^{\Theta}(S) &= \overline{\llbracket \diamond \rrbracket}^{\Theta}(\overline{\llbracket e_i \rrbracket}^{\Theta}(S)) && \text{by 6.4} \\ &\in \overline{\llbracket \diamond \rrbracket}^{\Theta}(\gamma_v(\mathcal{E}(e_i))) && \text{by 6.5} \\ &\subseteq \gamma_v(\mathbf{fst}(F_{\diamond}^{\#}(\mathcal{E}(e_i)))) && \text{by 6.2} \end{aligned}$$

Finally, by 6.1, this result and the soundness of the meet of value abstractions ensure that, if the concrete evaluation does not fail, then:

$$\begin{aligned} \forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \\ \overline{\llbracket \diamond(\vec{e}_i) \rrbracket}^{\Theta}(S) &\subseteq \gamma_v(\mathbf{fst}(F_{\diamond}^{\#}(\mathcal{E}(e_i)))) \cap \gamma_v(F_{\mathbb{D}}^{\#}(D, \diamond(\vec{e}_i))) \\ &\subseteq \gamma_v(\mathbf{fst}(F_{\diamond}^{\#}(\mathcal{E}(e_i)))) \sqcap_v F_{\mathbb{D}}^{\#}(D, \diamond(\vec{e}_i)) \\ &\subseteq \gamma_v(\mathcal{F}_{\diamond}^{\#}(\diamond(\vec{e}_i))) \end{aligned}$$

Therefore, the value abstraction $\mathcal{F}_{\diamond}^{\#}(\diamond(\vec{e}_i))$ used to update the valuation in the case of the forward evaluation of $\diamond(\vec{e}_i)$ is a sound approximation of the concrete evaluation of the expression e , if it does not fail. \square

Theorem 5 (Soundness of atomic updates). *In an abstract state D , an atomic step forward(\mathcal{E}, e) or backward(\mathcal{E}, e) produces a valuation \mathcal{E}' such that:*

$$\mathcal{E}' \sqsubseteq_{\mathbb{E}} \mathcal{E}$$

$$\forall S \in \mathcal{S} \text{ such that } \overline{\llbracket e \rrbracket}^{\Theta}(S) \neq \Omega, \quad S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}) \Leftrightarrow S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}')$$

Proof. Let \mathcal{E} be an initial valuation, D an abstract state, e an expression and v a value abstraction. The first condition of Theorem 5 is guaranteed by the update function:

$$\begin{cases} \forall e' \neq e, \text{update}(\mathcal{E}, e, v)(e') = \mathcal{E}(e') \\ e \in \text{dom}(\mathcal{E}) \Rightarrow \text{update}(\mathcal{E}, e, v)(e) = \mathcal{E}(e) \sqcap_v v \sqsubseteq_v \mathcal{E}(e) \end{cases}$$

We write $\mathcal{E}' = \text{update}(\mathcal{E}, e, v)$, and we have:

$$\forall e' \in \text{dom}(\mathcal{E}), e' \in \text{dom}(\mathcal{E}') \wedge \mathcal{E}'(e') \sqsubseteq_v \mathcal{E}(e') \Rightarrow \mathcal{E}' \sqsubseteq_{\mathbb{E}} \mathcal{E}$$

By lemma 19, this ensures that $\gamma_{\mathbb{E}}(\mathcal{E}') \subseteq \gamma_{\mathbb{E}}(\mathcal{E})$, and thus that $\gamma_{\mathbb{DE}}(D, \mathcal{E}') \subseteq \gamma_{\mathbb{DE}}(D, \mathcal{E})$, which implies the left-to-right implication of the second condition.

The converse stems from lemma 22. We assume $\llbracket e \rrbracket^{\Theta}(S) \neq \Omega$. Let S be a concrete state in $\gamma_{\mathbb{DE}}(D, \mathcal{E})$. As v is the value used to update the abstraction of the expression e in the valuation \mathcal{E} , the lemma ensures that:

$$\llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(v) \quad (6.6)$$

By definition 46, we have:

$$\forall e' \in \text{dom}(\mathcal{E}), \llbracket e' \rrbracket^{\Theta}(S) \in \gamma_v(\mathcal{E}(e')) \quad (6.7)$$

As $\forall e' \neq e, \mathcal{E}'(e') = \mathcal{E}(e')$, we deduce:

$$\forall e' \in \text{dom}(\mathcal{E}), e \neq e' \Rightarrow \llbracket e' \rrbracket^{\Theta}(S) \in \gamma_v(\mathcal{E}'(e'))$$

For the expression e itself, $\mathcal{E}'(e) = \mathcal{E}(e) \sqcap_v v$. Then:

- either $e \in \text{dom}(\mathcal{E})$, and by 6.7: $\llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(\mathcal{E}(e))$. Thus:

$$\llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(v) \cap \gamma_v(\mathcal{E}(e)) \subseteq \gamma_v(v \sqcap_v \mathcal{E}(e)) = \gamma_v(\mathcal{E}'(e))$$

- or $e \notin \text{dom}(\mathcal{E})$, and $\mathcal{E}'(e) = v \sqcap_v \top = v \ni \llbracket e \rrbracket^{\Theta}(S)$.

In both cases, as $\text{dom}(\mathcal{E}') = \text{dom}(\mathcal{E}) \cup \{e\}$, we finally have

$$\forall e \in \text{dom}(\mathcal{E}'), \llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(\mathcal{E}'(e))$$

which implies $S \in \gamma_{\mathbb{E}}(\mathcal{E}')$.

We have proved that for all concrete states S such that the evaluation of e succeeds,

$$S \in \gamma_{\mathbb{DE}}(D, \mathcal{E}) \Rightarrow S \in \gamma_{\mathbb{DE}}(D, \mathcal{E}')$$

□

6.1.6 Complete Evaluations

Complete evaluations of an expression e can now be defined by series of atomic steps. Thanks to the invariant of theorem 5, any sequence of the atomic steps on some subterms of e leads to a valuation more precise than the initial one, and which is a sound abstraction of the considered concrete states. But in order for the evaluation to be meaningful, these atomic updates must also be properly ordered. As defined previously, a forward evaluation is a bottom-up propagation, and performs the forward steps from the constants to the root expression, going up the operations. In particular, this ensures that the abstractions needed at one step have already been computed at a previous step, and are thus available in the valuation. On the other hand, the backward semantics of an operator need abstractions of its result and of its operands at the same time. Thus, a backward evaluation should follow a forward evaluation that gives a first value abstraction to each subterm of the root expression. Then, the valuation is updated with the new value for the expression, which is propagated through a sequence of backward steps, going down the operators until reaching constants.

Definition 49 (Forward evaluation). A possible definition of the forward evaluation $\llbracket e \rrbracket_D^\#(\mathcal{E})$ for an expression e in an abstract state D is:

$$\begin{aligned}
\llbracket c \rrbracket_D^\#(\mathcal{E}) &\triangleq \perp_{\mathbb{A}}, \text{forward}(\mathcal{E}, c) \\
\llbracket *_\tau a \rrbracket_D^\#(\mathcal{E}) &\triangleq \text{snd}(\mathbb{F}_{*_\tau}^\#(\mathcal{E}'(a))) \sqcup_{\mathbb{A}} \mathbf{A}', \text{forward}(\mathcal{E}', e) \\
&\quad \text{where } \llbracket a \rrbracket_D^\#(\mathcal{E}) = \mathbf{A}', \mathcal{E}' \\
\llbracket \diamond \vec{e}_i \rrbracket_D^\#(\mathcal{E}) &\triangleq \text{snd}(\mathbb{F}_{\mathbb{D}}^\#(D, \diamond \vec{e}_i)) \sqcap_{\mathbb{A}} (\text{snd}(\mathbb{F}_{\diamond}^\#(\overline{\mathcal{E}'(e_i)}))) \sqcup_{\mathbb{A}} \mathbf{A}', \\
&\quad \text{forward}(\mathcal{E}', e) \\
&\quad \text{where } \begin{cases} \llbracket e_1 \rrbracket_D^\#(\mathcal{E}) &= \mathbf{A}_1, \mathcal{E}_1 \\ \llbracket e_2 \rrbracket_D^\#(\mathcal{E}_1) &= \mathbf{A}_2, \mathcal{E}_2 \\ \dots & \\ \llbracket e_n \rrbracket_D^\#(\mathcal{E}_{n-1}) &= \mathbf{A}_n, \mathcal{E}' \\ \mathbf{A}' &= \mathbf{A}_1 \sqcup_{\mathbb{A}} \mathbf{A}_2 \dots \sqcup_{\mathbb{A}} \mathbf{A}_n \end{cases}
\end{aligned}$$

Theorem 6. The above definition of forward evaluation satisfies the soundness and precision properties stated in propositions 1 and 3.

Proof. Let e be an expression, \mathcal{E} an initial valuation and D an abstract state. Let S be a state in the concretization $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$. We write $\llbracket e \rrbracket_D^\#(\mathcal{E}) = \mathbf{A}, \mathcal{E}'$. The soundness and precision requirements on the valuation \mathcal{E}' is a direct consequence of theorem 5: as the result of applications of atomic steps, \mathcal{E}' satisfies:

$$\begin{aligned}
\mathcal{E}' &\sqsubseteq_{\mathbb{E}} \mathcal{E} \\
\overline{\llbracket e \rrbracket}^\theta(S) \neq \Omega &\Rightarrow S \in \gamma_{\mathbb{E}}(\mathcal{E}')
\end{aligned}$$

\mathcal{E}' also contains a value abstraction for e and for each subterm of e , as a forward step has been applied to each.

Finally, we can prove that $\mathbf{A} \models_{\mathbf{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S)$ by an inductive reasoning on expressions. Let us assume that $e = \diamond \vec{e}_i$ —the same proof applies for a dereference—, and that the alarm maps \mathbf{A}_1 to \mathbf{A}_n produced by $\llbracket e_1 \rrbracket_D^{\#}(\mathcal{E})$ to $\llbracket e_n \rrbracket_D^{\#}(\mathcal{E})$ are sound abstraction of the concrete evaluation of e_1 to e_n . By definition 42, the alarm map \mathbf{A} produced by the forward semantics $F_{\diamond}^{\#}$ is a sound abstraction of the operation. By lemma 9, the join of $\mathbf{A}_1, \dots, \mathbf{A}_n$ and \mathbf{A} is a sound abstraction of the evaluation of $\diamond \vec{e}_i$. By soundness of the domain query $F_{\mathbb{D}}^{\#}$, the alarm map $\text{snd}(F_{\mathbb{D}}^{\#}(D, \diamond \vec{e}_i))$ is also a sound abstraction of the evaluation of $\diamond \vec{e}_i$. By lemma 8, the meet of both alarm maps is still a sound abstraction of this evaluation. \square

Definition 50 (Backward evaluation). A possible definition of the backward evaluation $\llbracket \overleftarrow{e} \rrbracket_D^{\#}(\mathcal{E}, v)$ of an expression e to a value abstraction v in an abstract state D is:

$$\begin{aligned} \llbracket \overleftarrow{e} \rrbracket_D^{\#}(\mathcal{E}) &\triangleq \mathcal{E} \\ \llbracket \overleftarrow{*_{\tau} a} \rrbracket_D^{\#}(\mathcal{E}) &\triangleq \llbracket \overleftarrow{a} \rrbracket_D^{\#}(\text{backward}(\mathcal{E}, *_{\tau} a)) \\ \llbracket \overleftarrow{\diamond \vec{e}_i} \rrbracket_D^{\#}(\mathcal{E}) &\triangleq (\llbracket \overleftarrow{e_n} \rrbracket_D^{\#} \circ \llbracket \overleftarrow{e_{n-1}} \rrbracket_D^{\#} \circ \dots \circ \llbracket \overleftarrow{e_1} \rrbracket_D^{\#})(\text{backward}(\mathcal{E}, \diamond \vec{e}_i)) \\ \llbracket \overleftarrow{e} \rrbracket_D^{\#}(\mathcal{E}, v) &\triangleq \llbracket \overleftarrow{e} \rrbracket_D^{\#}(\text{update}(\mathcal{E}, e, v)) \end{aligned}$$

Theorem 7. The above definition of backward evaluation satisfies the soundness and precision properties stated in propositions 2 and 3.

Proof. This theorem results from theorem 5. Let e be an expression, v a value abstraction, \mathcal{E} a valuation and D an abstract state. We write:

$$\begin{aligned} \mathcal{E}' &= \text{update}(\mathcal{E}, e, v) \\ \mathcal{E}'' &= \llbracket \overleftarrow{e} \rrbracket_D^{\#}(\mathcal{E}') = \llbracket \overleftarrow{e} \rrbracket_D^{\#}(\mathcal{E}, v) \end{aligned}$$

Let S be a concrete state in $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$ such that $\overline{\llbracket e \rrbracket}^{\Theta}(S) \in \gamma_v(v)$. Then $\overline{\llbracket e \rrbracket}^{\Theta}(S) \in \gamma_v(v) \cap \gamma_v(\mathcal{E}(e))$ since $S \in \gamma_{\mathbb{E}}(\mathcal{E})$. We have:

$$\begin{aligned} \overline{\llbracket e \rrbracket}^{\Theta}(S) &\in \gamma_v(v) \cap \gamma_v(\mathcal{E}(e)) \subseteq \gamma_v(v \sqcap_v \mathcal{E}(e)) = \gamma_v(\mathcal{E}'(e)) \\ \forall e' \neq e, \overline{\llbracket e' \rrbracket}^{\Theta}(S) &\in \gamma_{\mathbb{E}}(\mathcal{E}(e')) = \gamma(\mathcal{E}'(e')) \end{aligned}$$

And thus, $S \in \gamma_{\mathbb{E}}(\mathcal{E}')$.

We have $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}')$ and $\overline{\llbracket e \rrbracket}^{\Theta}(S) \neq \Omega$. As \mathcal{E}'' results from a series of backward steps from \mathcal{E}' , theorem 5 implies that $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}'')$. Thus:

$$\forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \overline{\llbracket e \rrbracket}^{\Theta}(S) \in \gamma_v(v) \Rightarrow S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}'')$$

Moreover, we have $\mathcal{E}'' \sqsubseteq_{\mathbb{E}} \mathcal{E}' \sqsubseteq_{\mathbb{E}} \mathcal{E}$, and $\mathcal{E}''(e) \sqsubseteq_v \mathcal{E}'(e) \sqsubseteq_v v$. The conditions of proposition 3 are satisfied too. \square

6.1.7 *Simplified Implementation*

Figure 6.1 is a short OCaml implementation of a simple evaluator, following the foundations laid previously. Monads, introduced in Section 5.1.5, are widely used to handle both the bottom case and the alarms. The gradually updated valuation is an imperative reference. It is initially the empty map, unless otherwise specified.

6.1.7.1 *Forward Evaluator*

The forward evaluation naturally uses the valuation as a cache: if the valuation already contains a value abstraction for the requested expression, this abstraction is returned. Otherwise, a value abstraction is internally computed for the expression, added into the valuation, and then returned. As the forward evaluation also produces alarms, the valuation also caches the alarm maps gathered at each level of the evaluation of an expression. Note that the computation is done only if the given expression is missing in the valuation, so we can simply add the new abstractions in the valuation.

As explained in Section 6.1.2.2, an abstract domain may infer relevant information about arbitrary expressions. Thus, the cooperative forward evaluation involves the abstract domain systematically, which supplies it with sound abstractions (value and alarms) for the current expression. In parallel, the internal forward evaluation is the application of the abstract semantics corresponding to the expression. It uses the result of the main forward evaluation for the subterms. The propagation of the bottom case and the join of all alarms is hidden in the monad. Finally, the cooperative evaluation meets the abstractions given by the domain and those computed by the abstract semantics, as both are sound approximations of the evaluation.

In practice, the EVA evaluator only involves the abstract domain on some expressions: dereferences, cast, operators. These are the expressions on which a state abstraction may provide further precision than the internal evaluation. The constants and some particular constructs of the C language (as `sizeof`) are only internally processed by the value semantics. Also, the semantics of dereferences and the additional query on arbitrary expressions are kept separate in the interface of abstract domains.

6.1.7.2 *Backward Evaluator*

The backward evaluation starts with a new value abstraction for a given expression, and compares it with the value abstraction computed by a forward evaluation. If the new abstraction brings more precision, the meet between the former and the new value abstraction is stored in the valuation, and is then backward propagated to the subterm of the expression. If the former value abstraction is included

Listing 6.1: Implementation of a simplified evaluator

```

1  let (>>-) t f = match t with
    | 'Bottom -> Bottom
    | 'Value t -> f t

5  let (>>=) (t, a) f = match t with
    | 'Bottom -> 'Bottom, a
    | 'Value t -> let t', a' = f t in t', Alarmset.union a a'

let valuation = ref Valuation.empty

10 let rec forward_eval state expr =
    try Valuation.find !valuation expr
    with Not_found ->
        let r = coop_forward_eval state expr in
15     Valuation.add !valuation expr r; r

and coop_forward_eval state expr =
    let v, a = internal_forward_eval state expr in
    let v', a' = Domain.extract_expr state expr in
20     Value.meet v v', Alarms.meet a a'

and internal_forward_eval state = function
    | BinOp (op, e1, e2) ->
        forward_eval state e1 >>= fun v1 ->
25         forward_eval state e2 >>= fun v2 ->
            Value.forward_binop (e1, e2) op v1 v2
    | Const c -> Value.constant c
    | [...]

30 let rec backward_eval state expr new_value =
    let old_value, alarms = forward_eval state expr in
    if Value.is_included old_value new_value then 'Value ()
    else
        Value.meet new_value old_value >>- fun result ->
35         Valuation.add !valuation expr (result, alarms);
        internal_backward state result expr

and internal_backward state result = function
    | BinOp (op, e1, e2) ->
        fst (forward_eval state e1) >>- fun left ->
40         fst (forward_eval state e2) >>- fun right ->
            Value.backward_binop binop ~left ~right ~result
            >>- fun (v1, v2) ->
                backward_eval state e1 v1 >>- fun () ->
45         backward_eval state e2 v2
    | Const c -> 'Value ()
    | [...]

```

in the new abstraction or if the expression is a constant, the backward evaluation stops. Remember that the multiple calls to `forward_eval` amount to searching in the valuation, once the first forward evaluation has been done.

The backward evaluation directly updates the valuation. It returns the bottom case if it discovers a contradiction, or unit at the end of the propagation.

6.1.7.3 *Effective Implementation*

The implementation that Figure 6.1 presents is pleasantly simple, but also relatively naive. A real implementation would likely be more intricate; as an example, the generic evaluator of [EVA](#) now exceeds 1000 lines of code. Especially, Figure 6.1 omits the high-level evaluation functions that trigger each evaluation pass. Carefully choosing when to propagate information is a crucial matter in order to implement an efficient evaluation strategy. The next section presents the strategies used in [EVA](#) for the interweaving of forward and backward propagations.

6.2 FORWARD AND BACKWARD EVALUATION STRATEGIES

The last section presented the evaluation as a series of atomic updates of a valuation, following the abstract semantics provided by both value and state abstractions. Such updates preserve the required invariants of soundness and precision of the valuation. It now remains to determine an effective strategy for ordering these atomic operations that achieves a precise evaluation of a complete expression. As usual in abstract interpretation, this involves seeking a good trade-off between accuracy and efficiency. In particular, we will see in example 16 that applying an unbounded number of atomic updates may lead to a slow convergence before the valuation reaches a fixpoint.

As we have seen, starting from an empty valuation, the first step needs to be a forward propagation step from the constants of the expressions, as the backward semantics need value abstractions for the result of some operations. Also, applying successively the same semantics on the same subterm is pointless. Apart from that, any interweaving of forward and backward semantics could work.

However, there are a number of scenarios in which some propagations cannot bring new information in a valuation. In this section, we first focus on the conditions where a forward or a backward propagation step is useful, i.e. may improve further the precision of the current valuation. We then present the interweaving strategy between forward and backward propagations used in [EVA](#).

6.2.1 Backward Propagation of Reductions

We detail here the situations where an atomic propagation of the backward semantics is relevant, i.e. may compute more precise abstractions than a previous forward evaluation did. This is the case when alarms have been issued by the corresponding forward evaluation step, when an abstract domain improved the value abstraction given by the forward semantics, or for the conditional expression of an *if* statement.

6.2.1.1 Reduction on Alarms

Let us consider the application of a binary C operator \diamond . The same reasoning would apply for an operator of any arity. At the level of an expression $e = \diamond(e_1, e_2)$, the forward atomic step applies the forward semantics $F_\diamond^\#$ to some value abstractions v_1 and v_2 of e_1 and e_2 . It produces an alarm map \mathbf{A} and a value abstraction v of the result.

$$F_\diamond^\#(v_1, v_2) = (\mathbf{A}, v)$$

Then, a backward atomic step may try to reduce the abstractions v_1 and v_2 by the new abstractions v'_1 and v'_2 given by the backward semantics $B_\diamond^\#$.

$$B_\diamond^\#(v_1, v_2, v) = (v'_1, v'_2)$$

The soundness of the abstract semantics $B_\diamond^\#$ (definition 43) ensures that any concrete values V_1 and V_2 in the respective concretizations of v_1 and v_2 such that $\overline{[\diamond]}^\Theta(V_1, V_2) \in \gamma_v(v)$ are also in the respective concretizations of v'_1 and v'_2 .

$$\forall (V_1, V_2) \in \gamma_v(v_1, v_2), \quad \overline{[\diamond]}^\Theta(V_1, V_2) \in \gamma_v(v) \Rightarrow (V_1, V_2) \in \gamma_v(v'_1, v'_2)$$

Moreover, the soundness of the forward semantics $F_\diamond^\#$ (definition 42) ensures that for such concrete values $V_1 \in \gamma_v(v_1)$ and $V_2 \in \gamma_v(v_2)$, the concrete value $\overline{[\diamond]}^\Theta(V_1, V_2)$ is indeed in $\gamma_v(v)$, unless the operation fails with the error case. Thus, we have:

$$\forall (V_1, V_2) \in \gamma_v(v_1, v_2), \quad \overline{[\diamond]}^\Theta(V_1, V_2) \neq \Omega \Rightarrow (V_1, V_2) \in \gamma_v(v'_1, v'_2)$$

Finally, if the alarm map \mathbf{A} is the empty closed map (the bottom element representing the empty set of undesirable behavior), then the error case never happens.

$$\mathbf{A} = \text{none} \Rightarrow \gamma_v(v_1, v_2) \subseteq \gamma_v(v'_1, v'_2)$$

In this case, the new abstractions v'_1 and v'_2 computed by the backward abstract semantics cannot be more precise than the former abstractions v_1 and v_2 . The backward step is then pointless on the expression fragment e .

```

1 void f(char a, char b) {
    char w = 0;
    if (a > 49 && b > 49)
        w = a + b;
5 }

1 extern int t[10];
void g(int n) { int w = t[n] }

```

Figure 6.2: Backward propagation after alarm emission

Otherwise, if the alarm map \mathbf{A} is not empty, the application of the operator \diamond may fail for some concrete values V_1 and V_2 in the respective concretizations of v_1 and v_2 . Remember that an alarm map is only an over-approximation of undesirable behaviors, and it can be non empty even if no error occurs. Nevertheless, if the operation really fails for some concrete values, the backward semantics $B_{\diamond}^{\#}$ may successfully reduce the initial value abstractions v_1 and v_2 to remove such concrete values from their concretizations.

Example 11. Let us consider the first code of Figure 6.2, and more specifically the evaluation of the expression $a + b$ at line 4. All computations are done on the signed integer type of 1 byte. As the values of the variables a and b are unbounded to the right, the result of their addition may exceed the signed integer range, which is an undefined behavior. At this point, the forward semantics $F_{+}^{\#}$ of any value abstraction must produce a non empty alarm map, with at least the alarm $a+b \leq 127$ bound to an unknown status.

An interval analysis easily infers the interval $[50..127]$ as value abstraction for the variables a and b at line 4, and the interval arithmetic leads to the interval $[100..127]$ for $a+b$ (together with the alarm mentioned above). This resulting interval excludes some values for a and for b , those such $a+b \geq 128$. Indeed, as $a \geq 50$ and $b \geq 50$, we can deduce that both variables must be between 50 and 77. Thus, the backward propagation on intervals for this addition reduces the interval for a and b to $[50..77]$. Note that this interval would still not be sufficient to exclude the error case.

Example 12. On the second code of Figure 6.2, the dereference of $t[n]$ at line 5 raises an alarm, as the index n is unbounded. Then, with the cvalue representation of memory locations, the abstraction of the possible memory address of $t[n]$ is the map $\{\{ \&t+[0..9] \}\}$. The backward propagation of this abstraction allows reducing the possible value of n to the interval $[0..9]$.

Note that the same reductions could be achieved from the alarm map, which carries the same information about the inconsistent val-

ues of the operands of an operation. However, some backward propagations may also be relevant in the absence of alarms, when a value abstraction computed for an expression is directly reduced, as explained below.

6.2.1.2 Value Reduction on Conditional Statements

We have seen that on operations producing no alarms, a backward propagation of the value computed by the forward semantics is pointless: a sound backward semantics cannot achieve a better precision. However, the value abstraction computed by a forward evaluation can sometimes be reduced, and such a reduction may lead to further reductions downstream.

A natural place for a value reduction is on the condition of `if` statements, translated into $e == 0?$ filters in our language syntax. A $e == 0?$ filter halts the concrete states in which the expression e has not a zero value. Through a dataflow analysis, when an abstract state D reaches a filter $e == 0?$, its processing depends on the value abstraction computed by the forward evaluation of e :

- if the abstraction represents only a zero value, then the abstract state goes through the filter statement unchanged;
- if it is an abstraction of non-zero value only, the abstract state is not propagated through the filter statement;
- otherwise, the expression e may evaluate to zero and non-zero values. The abstract state must be propagated through the filter statement, but from this point forward, we can assume that the expression e evaluates to zero.

In the latter case, the valuation resulting from the forward evaluation of e is inaccurate. In this valuation, the value abstraction for the condition e can be replaced by an abstraction of the concrete zero value. Then, this more precise value for e can be backward propagated to the subterms of e , to make the valuation even more precise.

Example 13. Figure 6.3 illustrates the forward and backward evaluations of the condition of an `if` statement. The C code we consider is on top, next to its representation as a control-flow graph with our language syntax. The node `o` is the entry point of the function f , and the conditional construct is separated into two filter statements on the boolean expressions $x+y < 0$ and $x+y \geq 0$. The latter leads to the first branch, and the former to the second branch. The table below shows the valuations produced by the evaluations of each filter condition, with interval abstractions. The precondition of function f ensures that the variable x is between -10 and 10 , and the variable y is between 5 and 20 . The forward evaluation of the expression $x + y$ leads to the interval $[-5..30]$ in each edge, the constant 0 is evidently

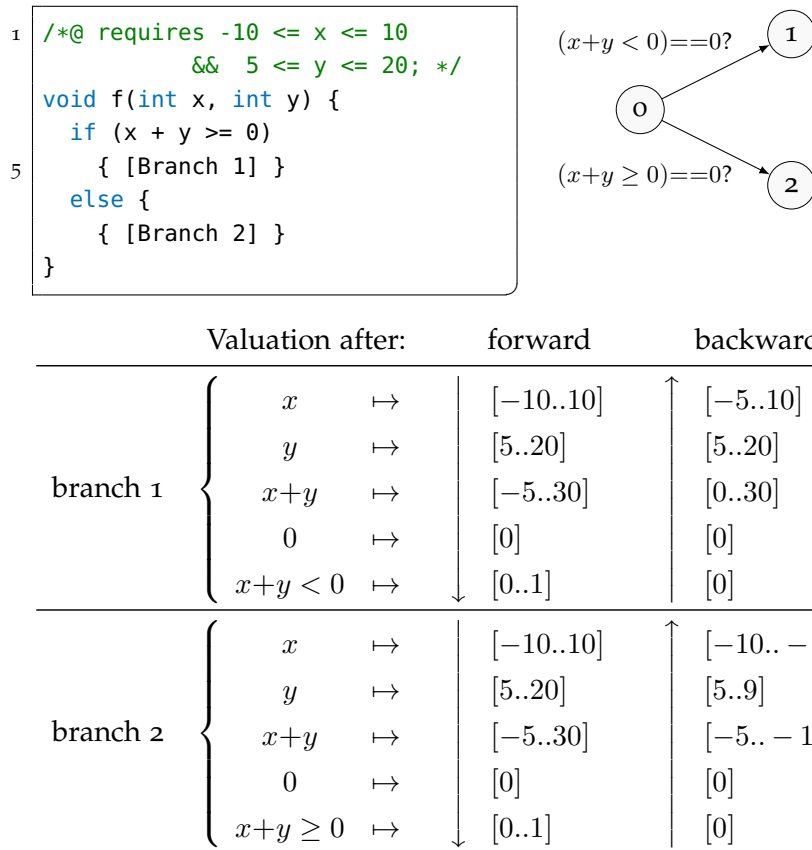


Figure 6.3: Backward propagation on a conditional statement

bound to the singleton interval $[0]$, and the truth value of each condition is unknown, represented by the interval $[0..1]$.

Then, in each edge, the value abstraction of the condition is reduced to the interval $[0]$, which is then backward propagated to the subterms. Note that in the table of Figure 6.3, the expressions are ordered following the forward evaluation; the backward one operates in the reverse order. The interval for constants cannot be reduced. In the first branch, $x+y < 0$ is equal to 0, so $x+y \geq 0$, and the interval for $x+y$ is reduced to $[0..30]$. This interval is again backward propagated, reducing x to $[-5..10]$ and leaving y unchanged. Conversely, in the valuation for second branch, $x+y$ is reduced to $[-5..-1]$, and then x is reduced to $[-10..-6]$ and y to $[5..9]$.

6.2.1.3 Value Reduction from the Abstract Domain

If the reductions on the conditionals of a program are especially natural, an abstract domain may also induce some reductions of the value abstractions computed during a forward evaluation of arbitrary expressions. Indeed, as explained in Section 6.1.2.2, a state abstraction provides not only the abstract semantics of dereferences, but also supplies the forward evaluator with value abstractions for arbitrary expressions. These value abstractions are based on the properties inferred by the domain, and may bring additional precision to the evaluation. In that case, it may also be beneficial to backward propagate them. Figure 6.4 presents an example where such a propagation may be useful.

Example 14. We focus on the last assignment of the left code of Figure 6.4, and more specifically on the evaluation of the expression $y-x$. We assume that this evaluation involves an interval domain and a relational domain, whose inferred properties at line 5 are shown at the right table of the figure. The interval bounds for x and y come directly from the assertion, and the inequality between them follows from the conditional. We also assume that the evaluation uses intervals as value abstractions. The internal forward evaluation uses the interval provided by the interval domain for the dereferences of the variables and computes $[-15..5]$ for the expression $y-x$. However, the relational domain can use its inequality to supply the interval $[1..+\infty]$ as a sound abstraction for this same expression. The meet of both intervals gives $[1..5]$, which is strictly more precise. Through the usual interval arithmetic, the backward propagation of this new value allows reducing the respective interval for x and y to $[0..4]$ and $[1..5]$, and thus makes the resulting valuation more accurate.

Ultimately, this backward propagation allows the interval domain to reduce the value abstraction it has inferred for the variables x and y . In practice, the reduction of the interval domain could also be done at the evaluation of the conditional, at line 4. However, when using

<pre> 1 /*@ assert 0 ≤ x ≤ 10 2 && -5 ≤ y ≤ 5; */ 3 b = x < y; 4 if (b) 5 w = y - x; </pre>	<table border="1"> <thead> <tr> <th>Intervals</th><th>Relational</th></tr> </thead> <tbody> <tr> <td>$\left\{ \begin{array}{l} x \mapsto [0..10] \\ y \mapsto [-5..5] \end{array} \right.$</td><td>$b = (x < y)$ $x < y$</td></tr> </tbody> </table>	Intervals	Relational	$\left\{ \begin{array}{l} x \mapsto [0..10] \\ y \mapsto [-5..5] \end{array} \right.$	$b = (x < y)$ $x < y$
Intervals	Relational				
$\left\{ \begin{array}{l} x \mapsto [0..10] \\ y \mapsto [-5..5] \end{array} \right.$	$b = (x < y)$ $x < y$				

Figure 6.4: Backward propagation from an abstract domain

deeply relational abstractions, achieving a maximal inter-reduction between abstract domains can be overly costly on intricate conditionals. In some cases, a better strategy is to postpone some reductions to the program points where the involved variables are actually used.

6.2.2 Forward Propagation of Reductions

A backward propagation aims at reducing some value abstractions of the valuation, increasing its precision. The same objective can also motivate a forward propagation. After a reduction achieved by a backward propagation, may a forward propagation improve the precision further?

6.2.2.1 Exact Forward Operator

We say that an abstract forward semantics $F_{\diamond}^{\#}$ is *exact* when they do not introduce irrelevant concrete values in the concretization of the resulting abstraction. This means that:

$$\forall (v_1, v_2) \in (\mathbb{V}^{\#})^2, \quad \forall V \in \gamma_v(F_{\diamond}^{\#}(v_1, v_2)), \exists (V_1, V_2) \in \gamma_v(v_1, v_2), V = \overline{[\diamond]}^{\theta}(V_1, V_2)$$

For instance, this is the case for the interval arithmetic of the integer addition and subtraction. Indeed, the abstract addition of interval abstractions is:

$$[i_1, s_1] +^{\#} [i_2, s_2] \triangleq [i_1 + i_2, s_1 + s_2]$$

and we have indeed:

$$\forall n \in [i_1 + i_2, s_1 + s_2], \exists n_1 \in [i_1, s_1] \wedge n_2 \in [i_2, s_2], n = n_1 + n_2$$

However, the abstract multiplication of intervals is not exact. As a simple example, $[0..4] *^{\#} [3] = [0..12]$, but there is no integer n between 0 and 4 such that $3 * n = 1$. Note that the abstract semantics of integer interval with congruence, used in the cvalue abstraction of EVA and described at Section 5.3, is exact on the multiplication with a constant, but not on any arbitrary multiplication.

6.2.2.2 Forward Propagation after a Backward Evaluation Step

We consider again the application of a binary operator \diamond , through the forward and backward evaluation steps of an expression $e = \diamond(e_1, e_2)$. We denote respectively by v_1 and v_2 the initial value abstractions of the operands e_1 and e_2 —and by v the value abstraction given by the forward abstract operator $F_{\diamond}^{\#}$ for the expression e .

$$F_{\diamond}^{\#}(v_1, v_2) = v$$

We then assume that a reduction has soundly narrowed this value abstraction v to v' . The former abstraction v' has to be at least as precise as the latter one v , but they may also be equal. Then, a backward step has reduced the value abstractions of the operands e_1 and e_2 to v'_1 and v'_2 . We thus have:

$$\begin{aligned} v' \sqsubseteq_v v \quad v'_1 \sqsubseteq_v v_1 \quad v'_2 \sqsubseteq_v v_2 \\ \forall (V_1, V_2) \in \gamma_v(v_1, v_2), \quad \overline{[\diamond]}^{\Theta}(V_1, V_2) \in \gamma_v(v') \Rightarrow (V_1, V_2) \in \gamma_v(v'_1, v'_2) \end{aligned}$$

Using these reduced abstractions, a forward step leads to a new value abstraction v'' for the expression e . We now have:

$$\forall (V_1, V_2) \in \gamma_v(v'_1, v'_2), \quad \overline{[\diamond]}^{\Theta}(V_1, V_2) \in \gamma_v(v'') + \Omega$$

If the forward abstract semantics for this operator are exact, then we also have:

$$\forall V \in \gamma_v(v), \exists (V_1, V_2) \in \gamma_v(v_1, v_2), V = \overline{[\diamond]}^{\Theta}(V_1, V_2)$$

As $v' \sqsubseteq_v v$, and from the soundness requirement of the backward step, we can deduce that:

$$\forall V \in \gamma_v(v'), \exists (V_1, V_2) \in \gamma_v(v_1, v_2), \begin{cases} V = \overline{[\diamond]}^{\Theta}(V_1, V_2) \\ (V_1, V_2) \in \gamma_v(v'_1, v'_2) \end{cases}$$

The second assertion implies that $\overline{[\diamond]}^{\Theta}(V_1, V_2) \in \gamma_v(v'')$, and thus $V \in \gamma_v(v'')$. Finally, we have that all concrete values of $\gamma_v(v')$ are also in $\gamma_v(v'')$, and thus the value abstraction v'' recomputed by the forward semantics cannot be more precise than the already reduced value abstraction v' for the expression $e = \diamond(e_1, e_2)$.

On the other hand, if the forward abstract semantics are not exact, the forward semantics may compute a more precise abstraction after a standard backward propagation. We give an example of such a situation, using the interval semantics of the multiplication.

Example 15. We consider the evaluation of the conditional $(3 * x) < 12$ of a filter statement, where the values of the variable x may range from 0 to 4. This corresponds to the evaluation of the conditional at

```

1  int x = -1;
   while (x < 4) {
       x++;
       if (3*x < 12)
5      [...]
   }

```

Figure 6.5: Forward propagation after a backward reduction

line 4 of the sample code shown on Figure 6.5. With interval abstractions, the initial forward evaluation naturally leads to no alarm and this valuation (we omit the constants):

$$\left\{ \begin{array}{ll} x & \mapsto [0..4] \\ 3 * x & \mapsto [0..12] \\ (3 * x) < 12 & \mapsto [0..1] \end{array} \right.$$

Then, a backward evaluation propagates the fact that the condition holds, and we obtain the updated, more precise valuation:

$$\left\{ \begin{array}{ll} x & \mapsto [0..3] \\ 3 * x & \mapsto [0..11] \\ (3 * x) < 12 & \mapsto [1] \end{array} \right.$$

Finally, a second forward evaluation makes the valuation even more precise, as $3 *^{\#} [0..3] = [0..9]$:

$$\left\{ \begin{array}{ll} x & \mapsto [0..3] \\ 3 * x & \mapsto [0..9] \\ (3 * x) < 12 & \mapsto [1] \end{array} \right.$$

At this point, neither forward nor backward propagation may reduce further the value abstraction computed in the valuation.

The second forward propagation is able to be more precise than the previous backward propagation because the interval arithmetic is not exact on the multiplication by a constant. On the other hand, the reduced product between intervals and congruences is exact on such an operation. With these abstractions, the value computed for the expression $3 * x$ by the first backward propagation is $[0..11]0\%3$, internally reduced to $[0..9]0\%3$. This is the most precise abstraction of the possible value of the expression, and thus cannot be reduced further. A second forward propagation is then pointless.

6.2.2.3 Forward Propagation after Multiple Reductions

When a subterm appears several times in an expression, a backward evaluation may lead to successive reductions of this subterm, whenever the propagation reaches it. Each of these reductions may have

```

1  /*@ assert 0 <= x <= 100; */
   if (x == x+1)
     [...]

```

		forward	backward	forward	backward	
x	\mapsto	[0..100]	[1..99]	[1..99]	[2..98]	
$x+1$	\mapsto	[1..101]	[1..100]	[2..100]	[2..99]	...
$x==x+1$	\mapsto	[0..1]	[1]	[0..1]	[1]	

Figure 6.6: Slow convergence of interval propagations

a different reason. In this case, a forward propagation may achieve further precision, even if the abstract semantics are exact.

Example 16. The processing of the unsatisfiable conditional ($x==x+1$) is a classical example of a slow convergence through the propagation of intervals. If the interval representing the possible values of the variable x is not a singleton, then the forward abstract semantics of interval are unable to prove that the condition does not hold. In the code of Figure 6.6, for example, the value of the variable x ranges from 0 to 100. Thus, the value of $x+1$ ranges from 1 to 101, and the interval abstractions cannot disprove the equality between both expressions. The backward propagation of this equality reduces the interval for each expression (including x) to the meet of both intervals, which is [1..100]. And the backward propagation of this new interval for $x+1$ reduces a second time the interval for the variable x , down to the interval [1..99]. Then, the cycle recurs: the forward propagation of these new intervals is still unable to disprove the equality, but reduces the value of $x+1$, and the backward propagation of the equality rules out the bounds of the interval for x . Ultimately, alternating forward and backward propagations leads to the empty interval, raising the bottom case. The table of Figure 6.6 shows the successive valuations produced by the first steps of this process. Here, proving the condition to not hold requires 50 iterations of a forward and a backward propagation of intervals.

6.2.3 Interweaving Forward and Backward Propagations

The two previous sections detail the cases where a forward or a backward propagation is relevant, i.e. may achieve further precision through the produced valuation. However, the last presented example points out that an unrestricted application of forward and backward propagation may lead to a slow convergence of the evaluation of an expression. Rather than always seeking for the most precise valuation for the given expression, a better balance between precision and efficiency has to be found in the use of both propagations.

This section expounds the evaluation strategies implemented in EVA for the interweaving of the forward and the backward propagations.

6.2.3.1 Complete Propagations Strategy

Section 6.1.5 presented the atomic steps of a forward or backward propagation. Each of these steps soundly updates the valuation, and they may thus be intertwined in any way that leads to the complete evaluation of a given expression. However, we choose to avoid mixing these forward and backward steps.

Instead, EVA always performs a complete propagation through an expression in a given direction: from the constants to the root expression for a forward evaluation, and conversely for a backward one. While opposite full propagations may follow one after another, there is no alternating during a complete propagation, and thus no direct interaction between forward and backward phases. This choice makes the implementation much simpler, as in the code presented in Figure 6.1. The forward and the backward mechanisms are kept separate, rather than being all mutually recursive functionalities. This design allows a better control of the evaluation convergence, as we seek for a single global fixpoint of the computed valuation. It also avoids successive reductions of the same subterm where the last one supersedes the others. The following example illustrates this.

Example 17. Let us consider the evaluation of the expression $2x - 1$, where the value of the variable x is unknown. We assume that the computations are done in the signed integer type of 1 byte, whose values range from -128 to 127 . As signed integer overflows are prohibited, the multiplication $2 * x$ requires the value of x to be in the interval $[-64..63]$. Furthermore, the next subtraction $2x - 1$ requires $2x$ to be in the interval $[-127..127]$, and thus x to be in $[-63..63]$.

If each atomic step that produces some alarms triggers a backward propagation of its resulting value, the value abstraction of variable x would be reduced twice in the valuation. Actually, the first backward propagation of the value $[-128..126]$ for the subterm $2x$ is useless, as the upcoming backward propagation of the value $[-128..125]$ for the expression $2x - 1$ is strictly stronger. We can also notice that if the variable x is replaced by a more complex expression, each backward propagation may be overly costly.

Thus, instead of performing a backward propagation whenever possible, the evaluator first completes the forward propagation, and only then starts a backward propagation of the computed value for the root expression $2x - 1$. On this example, this leads to the same precise valuation as the other strategy, but minimizes the number of backward steps done.

We argue that this strategy of complete propagations is always as precise as triggering opposite propagations as soon as possible. We call this latter strategy the greedy algorithm. Remember that during these propagations, the valuation contains the value abstractions previously computed for all the subterms of the evaluated expression. The atomic propagation steps always meet the new value abstraction they infer for some subterm with the one stored in the valuation. Thus, the valuation can only become more and more precise through the propagations. An atomic step always uses the value abstractions stored in the valuation. Now, during a (forward or backward) propagation, when an opposite atomic step is relevant on a subterm t , it is postponed until the end of the current propagation. Afterwards, the complete opposite (backward or forward) propagation reaches this subterm t , and makes the postponed step using the abstractions of the current valuation. These value abstractions are at least as precise as the ones that were in the valuation during the previous propagation phase. Thus, the postponed atomic step is at least as precise as the initial step would have been.

However, in some specific cases, the strategy of complete propagations may require more computations to achieve the same precision as the alternate strategy. Indeed, during a (forward or backward) propagation A , an opposite propagation B may reduce some abstractions that will be used by the current propagation A . Then, the A propagation leads to less precise results due to the lack of reductions, which the B propagations will make later. After the B propagation, another complete propagation A' will however lead to the same precision as the greedy algorithm. The following example illustrates this.

Example 18. Consider now the evaluation of the quite contrived expression $(1/x)+x$, with the variable x having a value in $\{0; 1\}$. The division by x requires x to have a non-zero value. A backward step at the subterm $1/x$ will reduce the value of x to $\{1\}$. Note that we use here integer sets as value abstractions.

If the expression is evaluated from left to right, the greedy algorithm produces the following successive valuations at each step:

valuation after	1	2	3	4
$\left\{ \begin{array}{l} x \mapsto \\ 1/x \mapsto \\ (1/x)+x \mapsto \end{array} \right.$	$\{0; 1\}$ \top \top	$\{0; 1\}$ $\{1\}$ \top	$\{1\}$ $\{1\}$ \top	$\{1\}$ $\{1\}$ $\{2\}$

1. forward evaluation of the variable x (for the evaluation of $1/x$)
2. forward evaluation of $1/x$
3. backward propagation of the value $\{1\}$ for $1/x$
4. forward evaluation of $(1/x)+x$

On the other hand, our strategy of complete propagations postpones any backward step after the first forward evaluation, which does not benefit from the reduction of the variable x . Thus, the produced valuation are the following:

valuation after	1	2	3
$\left\{ \begin{array}{l} x \mapsto \\ 1/x \mapsto \\ (1/x)+x \mapsto \end{array} \right.$	$\{0; 1\}$	$\{1\}$	$\{1\}$
	$\{1\}$	$\{1\}$	$\{1\}$
	$\{1; 2\}$	$\{1; 2\}$	$\{2\}$

1. first complete forward propagation
2. complete backward propagation
3. second complete forward propagation

Both strategies lead ultimately to the same valuation, but here, our strategy requires more computations. However, note that the number of steps for the greedy algorithm depends on the evaluation order of the subterm. If the right operand of the addition is processed before the left operand, an additional forward propagation is needed anyway to take into account the reduction of x .

6.2.3.2 Triggering Relevant Propagation Steps

The propagations are done globally on entire expressions. However, as explained in both previous sections, the atomic steps of a propagation are not all relevant: some are unable to improve the precision of the valuation resulting from a previous computation. We naturally want the propagations to avoid performing pointless operations, without forgetting any effective reduction. This requires keeping track of the subterms where a new propagation may be useful.

At each level of a complete (forward or backward) propagation, EVA checks whether an opposite propagation step may improve the precision further, according to the criteria listed in Sections 6.2.1 and 6.2.2. Through a forward propagation, these are the steps where alarms are produced, or where an abstract domain reduces the abstraction internally computed by the value semantics. Through a backward propagation, this only requires the reduction of the value abstraction stored in the valuation. The information is internally stored in the valuation —this information is not exported in its interface. Then, a (forward or backward) propagation goes through the entire expression, but uses this information to only perform the relevant atomic steps. A propagation always needs to cross the whole expression, to ensure not to neglect some possible reductions. Through the propagation, an atomic step of the abstract semantics is performed if and only if:

- this propagation is marked as relevant in the valuation;

- the current propagation has reduced some value abstractions on which this semantics step is based (compared to the abstractions previously computed and stored in the valuation).

Example 19. Let us consider the expression $((3x+5)/4) - 1$, where x is a signed integer of 1 byte. We assume known that the variable x ranges from 0 to 42. With the interval abstractions, a complete forward propagation leads to the following valuation. The exclamation mark indicates the relevance of a backward propagation for a subterm (here because of the alarm preventing the addition to overflow).

$$\left\{ \begin{array}{ll} x & \mapsto [0..42] \\ 3x & \mapsto [0..126] \\ 3x+5 & \mapsto [5..127] \quad ! \\ (3x+5)/4 & \mapsto [1..31] \\ ((3x+5)/4) - 1 & \mapsto [0..30] \end{array} \right.$$

Then, a backward propagation omits the subtraction and the division step, but propagates backward the interval $[0..127]$ through the addition to its operands, using the backward semantics of interval. This reduces the interval for $3x$ to $[0..122]$. Thus, this new interval is again backward propagated through the multiplication. This reduces the interval for x to $[0..40]$. The resulting valuation is the following, where this time, the exclamation mark indicates the relevance of a *forward* propagation.

$$\left\{ \begin{array}{ll} x & \mapsto [0..40] \quad ! \\ 3x & \mapsto [0..122] \quad ! \\ 3x+5 & \mapsto [5..127] \\ (3x+5)/4 & \mapsto [1..31] \\ ((3x+5)/4) - 1 & \mapsto [0..30] \end{array} \right.$$

Then, a forward propagation may start again. As the interval abstraction of x has been reduced by the previous backward propagation, it is now forward propagated. The forward semantics of multiplication reduces the interval for $3x$ to $[0..120]$, which is in turn used by the forward semantics of addition and leads to the reduction of the interval for $3x+1$ to $[5..125]$, which is in turn used by the forward semantics of division, but leads to the same interval as before for $(3x+5)/4$. Thus, the propagation avoids the following step, and returns the valuation:

$$\left\{ \begin{array}{ll} x & \mapsto [0..40] \\ 3x & \mapsto [0..120] \\ 3x+5 & \mapsto [5..125] \\ (3x+5)/4 & \mapsto [1..31] \\ ((3x+5)/4) - 1 & \mapsto [0..30] \end{array} \right.$$

There was no alarm during this last forward propagation, and no domain was involved either. Thus, there is no place for a backward propagation, and the evaluation should stop here: this is the most precise valuation we can obtain for the evaluation of the expression $((3x+5)/4) - 1$, knowing that the variable x has a value in the interval $[0..42]$.

6.2.3.3 *The Interweaving Strategy*

This section finally explains how the complete forward and backward propagations are intertwined. As shown by example 16, the unbounded alternating of both kinds of propagation may take a large number of iterations until reaching a fixed point, namely a valuation that cannot be improved by any new propagation. Instead, we use a fixed number of iterations: an initial forward evaluation, then a complete backward propagation, and finally a second complete forward propagation for conditional expressions only. These complete propagations are unconditionally made, but as explained in the previous subsection, they only perform the relevant atomic steps. This strategy spares the evaluator from dealing with slow convergence issues, and has little impact on the precision of the resulting valuations, as shown by our experiments related in Section 8.1.3.1.

INITIAL FORWARD EVALUATION In any context, the evaluation of an expression starts necessarily by a complete forward propagation. This fills the valuation with a value abstraction for each subterm of the expression, and gathers the alarms needed to exclude the undesirable behaviors that the evaluation may cause. If the starting valuation is non-empty, the evaluation uses it to avoid the computation of the expressions it contains.

BACKWARD PROPAGATION Then, a complete backward propagation tries to improve the precision of the valuation on the subterms of the expression. For the conditional expression of a test statement, if the computed abstraction for the expression represents non-zero concrete values, it is reduced to the zero abstraction and marked as relevant for the backward propagation. If the computed abstraction represents only non-zero values, the condition is unsatisfied, and the evaluation raises the bottom case.

For arbitrary expressions, this backward propagation only aims at reducing some subterms according to the emitted alarms or to the contributions of the abstract domain. It is especially useful to reduce the value abstraction of variables, as the main abstract domain of EVA is a memory model of variables: it will learn only from the abstractions computed for variables. Backward reductions may also avoid the emission of successive redundant alarms, as in the next example.

Example 20. On the code of Listing 6.2, an alarm has to be emitted at line 2 to prevent the addition from overflowing. Without any reduction of the variable i , the same alarm would be emitted at line 3 as well. The backward propagation of the value abstraction computed for $i+1$ to the operand i may solve this issue.

Listing 6.2: Avoiding redundant alarms

```
1 void main (char i) {
  int a = i+1;
  int b = i+1;
}
```

SECOND FORWARD EVALUATION FOR CONDITIONALS For arbitrary expressions involved in assignments, **EVA** does only the initial forward evaluation and one complete backward propagation. However, on the conditional expression of a test statement, **EVA** performs a second forward propagation. Indeed, the backward propagation of conditions is especially important, as it reflects that the condition holds after the test statement. Moreover, this second forward propagation aims not only at reducing even more the value abstractions computed by both previous passes, but it also:

- determines whether the last backward propagation has fully reduced the subterms according to the truth value of the condition;
- reveals the inconsistency of the condition in some rare cases.

If the second forward propagation evaluates the condition as true, then no other reduction is needed to take into account that the condition holds. We then say that the backward propagation has fully reduced the subterms of the condition. Otherwise, an imprecise value abstraction for the condition means that one backward pass was not enough to fully propagate the truth value of the condition on its subterms (and especially its variables). As in Example 16, more propagations may solve the issue, at the expense of the efficiency of the evaluation. Instead, **EVA** stops the propagations here, but enables another mechanism to recover some precision: the partitioning of the evaluation, described in the following subsection.

Finally, the second forward evaluation may also discover an inconsistency. The following example presents such a case.

Example 21. In any execution of the code presented on the Listing 6.3, the condition at line 4 does not hold. However, the semantics of intervals is unable to prove this fact on a single forward propagation: as the precise value of the variable i is unknown, $t[i]$ and $t[i+1]$ are

Listing 6.3: Second forward propagation on conditional expression

```

1  int t[4] = {0, 1, 2, 3};
   /*@ requires 0 <= i <= 2; */
   void main (int i) {
       if (t[i] == t[i+1])
5      [...]
   }

```

respectively depicted by the intervals $[0..2]$ and $[1..3]$, whose intersection is not empty. Then, a backward propagation assumes that the conditional expression evaluates to true, and reduces the interval for both expressions to the meet $[1..2]$. To the left, this allows reducing the variable i to $[1..2]$; to the right, this allows reducing i to $[0..1]$. At the end of the backward propagation, i is bound to the singleton interval $[1]$ in the valuation.

Stopping the evaluation here would lead to a quite odd valuation, where the variable i has the precise value $[1]$ for which the condition is clearly false, while the condition is still considered as true. The second forward propagation solves this issue, raising the bottom case when reaching the equality.

6.2.4 Evaluation Subdivision

6.2.4.1 General Considerations

An evaluation can be split according to the possible values of a sub-term. Let us consider the evaluation of an expression e , and a sub-term t of e . Given a partition $\{v_1, \dots, v_n\}$ of the possible values of t , the evaluation is done independently for each v_i , assuming t has a value in v_i . The result (alarm map and valuation) of each evaluation are then joined. This mechanism was introduced to mitigate the absence of relational information in the former **VALUE** abstract interpreter. It has been maintained in **EVA** with some improvements. The following example illustrates how subdividing an evaluation may improve its precision.

Example 22. This example uses the standard interval arithmetic as abstract value semantics. We first consider the evaluation of the expression $x*x$ where x has a value between -10 and 10 . The evaluation of $x*x$ naturally leads to the interval $[-100..100]$. However, when $x \in [-10..0]$ or when $x \in [0..10]$, the evaluation leads to the interval $[0..100]$. We can conclude that $[0..100]$ is a sound approximation of $x*x$ when $x \in [-10..10]$.

Listing 6.4: Subdivision of evaluation

```

1  int t[5] = {0, 1, 2, 3, 4};
   /*@ assert 0 <= i <= 4; */
   if (t[i] == i) {
       [...]
5  }

```

Listing 6.4 presents another example of useful subdivision. When evaluating the condition at line 3, both expressions $t[i]$ and i have a value in the interval $[0..4]$, but the interval semantics cannot guarantee that they have the same value in this interval. Dividing the evaluation according to each possible value of i proves the equality in each case. Obviously, this technique is only reasonable when the interval for i is small enough.

6.2.4.2 Formalization

The soundness of subdividing an evaluation is stated by the following theorem.

Theorem 8 (Subdivision of a forward evaluation). *Let e be an expression, \mathcal{E} be a valuation, and D be an abstract state. If $\mathcal{E}_1, \mathcal{E}_2, \dots, \mathcal{E}_n$ is a set of valuations such that*

$$\begin{cases} \forall i \in \{1, \dots, n\}, \mathcal{E}_i \sqsubseteq_{\mathbb{E}} \mathcal{E} \\ \gamma_{\mathbb{E}}(\mathcal{E}) = \gamma_{\mathbb{E}}(\mathcal{E}_1) \cup \gamma_{\mathbb{E}}(\mathcal{E}_2) \cup \dots \cup \gamma_{\mathbb{E}}(\mathcal{E}_n) \end{cases}$$

then the subdivision of an evaluation according to the sets $\{\mathcal{E}_i\}$, defined as:

$$\llbracket e \rrbracket_D^{\#}(\mathcal{E}) = \bigsqcup_{i \in \{1 \dots n\}} (\llbracket e \rrbracket_D^{\#}(\mathcal{E}_i))$$

satisfies the properties 1 and 3 of a forward evaluation in \mathcal{E} . (Here, \bigsqcup is the join of pairs of valuations and alarm maps.)

It is worth noting that the theorem condition $\gamma_{\mathbb{E}}(\mathcal{E}) = \cup_i(\gamma_{\mathbb{E}}(\mathcal{E}_i))$ is stronger than the condition $\mathcal{E} = \sqcup_i(\mathcal{E}_i)$, which only guarantees that $\gamma_{\mathbb{E}}(\mathcal{E}) \supseteq \cup_i(\gamma_{\mathbb{E}}(\mathcal{E}_i))$.

Proof. We use the same notations as the theorem. Let S be a concrete state in $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$. As $\gamma_{\mathbb{E}}(\mathcal{E}) = \cup_i(\gamma_{\mathbb{E}}(\mathcal{E}_i))$, there exists i such that $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}_i)$.

Let $\mathcal{E}', \mathbf{A} = \llbracket e \rrbracket_D^{\#}(\mathcal{E}_i)$. By proposition 1:

$$\begin{cases} \mathbf{A} \models_{\mathbb{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S) \\ \overline{\llbracket e \rrbracket}^{\Theta}(S) \neq \Omega \Rightarrow S \in \gamma_{\mathbb{E}}(\mathcal{E}') \end{cases}$$

Let $\mathcal{E}'', \mathbf{A}'' = \bigsqcup_{i \in \{1 \dots n\}} (\llbracket e \rrbracket_D^\#(\mathcal{E}_i))$. We have $\mathcal{E}' \sqsubseteq_{\mathbb{E}} \mathcal{E}''$ and $\mathbf{A} \sqsubseteq_{\mathbf{A}} \mathbf{A}''$, which implies:

$$\begin{cases} \mathbf{A}'' \models_{\mathbf{A}} \overline{\llbracket e \rrbracket}^\Theta(S) \\ \overline{\llbracket e \rrbracket}^\Theta(S) \neq \Omega \Rightarrow S \in \gamma_{\mathbb{E}}(\mathcal{E}'') \end{cases}$$

This holds for any concrete state S in $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$, validating proposition 1.

Let $(\mathcal{E}'_i, \mathbf{A}_i) = \llbracket e \rrbracket_D^\#(\mathcal{E}_i)$. Proposition 3 ensures that $\mathcal{E}'_i \sqsubseteq_{\mathbb{E}} \mathcal{E}_i$ and $e \in \text{dom}(\mathcal{E}'_i)$. By hypothesis, $\mathcal{E}_i \sqsubseteq_{\mathbb{E}} \mathcal{E}$. Finally:

$$\forall i \in \{1, \dots, n\}, \begin{cases} e \in \text{dom}(\mathcal{E}'_i) \\ \mathcal{E}'_i \sqsubseteq_{\mathbb{E}} \mathcal{E} \end{cases} \Rightarrow \begin{cases} e \in \text{dom}(\bigsqcup_i(\mathcal{E}'_i)) \\ \bigsqcup_i(\mathcal{E}'_i) \sqsubseteq_{\mathbb{E}} \mathcal{E} \end{cases}$$

This validates proposition 3 for $\bigsqcup_i(\mathcal{E}'_i)$. \square

6.2.4.3 Heuristics and Implementation

Theorem 8 provides a sound and general context for subdividing an evaluation. However, repeating an evaluation in n different valuations has a high cost—it basically multiplies the evaluation time by n . The use of the valuation as a cache alleviates lightly the cost of a subdivision: only the computations that actually depend on the split subterm have to be performed again.

Example 23. We consider the expression $((x+y)*(x+y)+ e)$. If the sign of $x+y$ is not known, subdividing the evaluation according to this subterm is useful, as shown by the previous example. However, there is no need to repeat the computation of the addition $x+y$ or the evaluation of e . In fact, the subdivision can be limited to the subterm $(x+y)*(x+y)$. The resulting valuations are then joined, and the evaluation of the complete expression can continue.

While theorem 8 is proved for any “partition” of a valuation, the evaluation subdivision in EVA is limited to the partitioning of the abstract value of only one subterm of the considered expression. From a current valuation \mathcal{E} and for a subterm t , we assume given a set of abstract value v_1, \dots, v_n such that:

$$\begin{cases} \forall i \in \{1, \dots, n\}, v_i \sqsubseteq_{\mathbb{E}} \mathcal{E}(t) \\ \gamma_v(\mathcal{E}(t)) = \gamma_v(v_1) \cup \gamma_v(v_2) \cup \dots \cup \gamma_v(v_n) \end{cases} \quad (6.8)$$

Then, the set of valuations $\mathcal{E}_i = \mathcal{E}[t \rightarrow v_i]$ satisfies the condition of theorem 8, and the evaluation subdivision according to them is sound.

From an arbitrary abstract value $\mathcal{E}(t)$, there is not always a set of values satisfying equation 6.8. As example, a garbled mix (see Section 5.3.2) cannot be divided into a partition of smaller abstractions.

On the other hand, an interval can be divided into many different partitions. In [EVA](#), the subdivision is done only on the intervals and the integer sets of the cvalue abstraction (see Section [5.3.1](#)), in compliance with the hypothesis of theorem [8](#). The value combination structuring, described in Section [3.3](#), is used to work on the cvalue component of an arbitrary value abstraction (if this component exists).

Finally, the evaluation engine of [EVA](#) triggers such a subdivision when:

- on the condition of an *if* statement, the second forward evaluation proves that the backward propagation failed to fully propagate the truth value of the condition on its subterms (see Section [6.2.3](#)).

In this case, a subdivision is applied on any integer variable:

- appearing in the address of a dereference (in an array offset or in the computation of a pointer);
- whose possible values are represented by an integer set.

The subdivision is done by considering separately all integers in the set. As the cardinal of integer sets are kept small by the analyzer, this does not overly impact the analysis time. This is exactly the instance described in listing [6.4](#).

- on the right expression e of an assignment that contains several occurrences of the same syntactic subterm t , if the possible values of t are approximated by a basic integer or floating cvalue. The subdivided evaluation is limited to the smaller subterm of e that contains all occurrences of t , as already illustrated in example [23](#). The number of divisions of an interval is specified by a parameter of the abstract interpreter.

Although a subdivision is limited to one subterm, successive subdivisions of different subterms can be performed on the evaluation of an expression. On the expression $(x*x + y*y)$, [EVA](#) executes two subdivisions:

- one of the subterm of $x*x$, according to the value of x ;
- one of the subterm of $y*y$, according to the value of y .

Part IV

ABSTRACT SEMANTICS OF STATEMENTS

STATE ABSTRACTIONS

An abstract domain \mathbb{D} is a collection of state abstractions carrying information about program variables. Each abstract state D represents a set of concrete states S at a program point. A concretization function $\gamma_{\mathbb{D}}$ links abstract states to the concrete states they represent.

$$\gamma_{\mathbb{D}} : \mathbb{D} \rightarrow \mathcal{P}(\mathcal{S})$$

A domain can be a combination of several domains $\mathbb{D}_1, \mathbb{D}_2, \dots, \mathbb{D}_n$. Its abstract states are then tuples (d_1, d_2, \dots, d_n) representing the intersection of the concretization of each subdomain:

$$\gamma_{\mathbb{D}}(d_1, d_2, \dots, d_n) = \gamma_{\mathbb{D}}(d_1) \cap \gamma_{\mathbb{D}}(d_2) \cap \dots \cap \gamma_{\mathbb{D}}(d_n)$$

This section expounds how the subdomains of a combination interact with each other in our architecture.

Abstract domains provide transformers implementing an abstract semantics of statements. Each transformer models the effect of a statement on abstract states, inferring properties about how a program execution may behave at this point. The abstract interpreter uses them to propagate abstract states through the [CFG](#) of a program. At a node of the [CFG](#), the computed abstract state is an over-approximation of the possible concrete states given by the concrete semantics of the language.

The above description of abstract domains is absolutely standard in abstract interpretation. This chapter does not develop the general notion of abstract domains further, but only focuses on the interactions between them. The previous chapter was dedicated to the evaluation of an expression into alarms and value abstractions. This evaluation is collaborative: it involves all available domains, extracting information from each abstract state and pooling them. This information is expressed with alarm maps and value abstractions. The abstract domains provide the mandatory abstract semantics for the dereference of lvalues, but they can also supply the evaluator with some abstractions of other expressions. To compute precise abstractions, an abstract domain may request additional information from other domains. Finally, an abstract domain can trigger the backward propagation of value abstractions for some expressions.

When processing a statement, the abstract interpreter evaluates the expressions involved in the statement. At the end of these evaluations, a valuation (see [Section 6.1.3](#)) stores all the produced alarms and value abstractions. They have been cooperatively computed using the properties inferred by each abstract domain at the program

Listing 7.1: Signature of domain queries

```

1  type 'a or_bottom = 'Bottom | 'Value of 'a
   type 'a evaluated = 'a or_bottom * alarms

   val extract_lval :                               (*  $F_{*\tau}^\#$  *)
5     oracle:(exp -> value evaluated) ->
       state -> lval -> typ -> location -> value evaluated

   val extract_expr :                               (*  $F_{\mathbb{D}}^\#$  *)
9     oracle:(exp -> value evaluated) ->
10    state -> exp -> (value * origin) evaluated

```

point before the statement. Then, these abstractions are available for the abstract domain transformers to model the semantics of the statement as precisely as possible. Thus, information may flow from a domain to another, through the shared evaluation of expressions.

7.1 COLLABORATION FOR THE EVALUATION: DOMAIN QUERIES

A state abstract domain \mathbb{D} provides two *query* functions, on which the generic evaluator relies. Those functions translate *AST* fragments into alarm maps and value abstractions. If the domain is a product of several domains, all their answers to a query are intersected thanks to the meet operator of the alarm and value lattices. Thereby, each domain may contribute to reduce the abstract value computed for an expression, or decrease the number of emitted alarms.

These queries have been briefly presented through Section 6.1.2, but this section illustrates them in examples. Listing 7.1 shows their OCaml signature. The explanation of the oracle argument is postponed to Subsection 7.1.3.

7.1.1 Semantics of Dereference

The first query is a forward abstract semantics $F_{*\tau}^\#$ for dereferences, mandatory to perform the evaluation of expressions.

Definition 51 (Semantics of dereference). In an abstract state D , a sound abstract semantics $F_{*\tau}^\#$ of dereferences is a function from a location abstraction l to a value abstraction v and a map of alarms \mathbf{A} such that:

- v is a sound approximation of the concrete values that may be stored at any concrete address of $\gamma_{\mathbb{L}}(l)$ in any concrete state of $\gamma_{\mathbb{D}}(D)$;

- **A** is a sound abstraction of the possible failure of the dereference of any concrete address in $\gamma_{\mathbb{L}}(l)$. Therefore, the assertions of **A** ensure that the dereference succeeds.

$$\begin{aligned} & \mathbf{F}_{*\tau}^{\#} : \mathbb{D} \rightarrow \mathbb{L}^{\#} \rightarrow \mathbb{V}^{\#} \times \mathbb{A} \\ & \forall a \in \gamma_v(l), \forall S \in \gamma_{\mathbb{D}}(D), \mathbf{F}_{*\tau}^{\#}(D, l) \models_{\mathbb{V} \times \mathbb{A}} \overline{\llbracket *_{\tau} a \rrbracket}^{\Theta}(S) \end{aligned}$$

The alarms produced by $\mathbf{F}_{*\tau}^{\#}$ ensure the validity of the location, and that the contents of the read memory slice are proper (i.e. not *indeterminate* in C parlance, that is not `uninit` or `none` in our semantics of `clike`). They have been described in Section 5.1.3.2.

Proposition 4. For a product of domains $\mathbb{D} = \mathbb{D}_1 \times \dots \times \mathbb{D}_n$, a sound abstract semantics $\mathbf{F}_{*\tau}^{\#}$ can be defined by:

$$\mathbf{F}_{*\tau}^{\#}((d_1, \dots, d_n), l) = \mathbf{F}_{*\tau}^{\#}(d_1, l) \sqcap_{\mathbb{V} \times \mathbb{A}} \dots \sqcap_{\mathbb{V} \times \mathbb{A}} \mathbf{F}_{*\tau}^{\#}(d_n, l)$$

Proof. Let l be a location abstraction and $D = (d_1, d_2, \dots, d_n)$ be a product of n abstract states from domains $\mathbb{D}_1, \mathbb{D}_2, \dots, \mathbb{D}_n$. For each domain \mathbb{D}_i , we have:

$$\forall S \in \gamma_{\mathbb{D}}(d_i), \mathbf{F}_{*\tau}^{\#}(d_i, l) \models_{\mathbb{V} \times \mathbb{A}} \overline{\llbracket *_{\tau} a \rrbracket}^{\Theta}(S)$$

We thus have:

$$\begin{aligned} S \in \gamma_{\mathbb{D}}(d_1, \dots, d_n) & \Rightarrow \forall i \in \{1, \dots, n\}, S \in \gamma_{\mathbb{D}}(d_i) \\ & \Rightarrow \forall i \in \{1, \dots, n\}, \mathbf{F}_{*\tau}^{\#}(d_i, l) \models_{\mathbb{V} \times \mathbb{A}} \overline{\llbracket *_{\tau} a \rrbracket}^{\Theta}(S) \\ & \Rightarrow \mathbf{F}_{*\tau}^{\#}((d_1, d_2, \dots, d_n), l) \models_{\mathbb{V} \times \mathbb{A}} \overline{\llbracket *_{\tau} a \rrbracket}^{\Theta}(S) \end{aligned}$$

The last deduction is the application of lemma 11. \square

Example 24. Figure 7.1 illustrates the collaboration between state abstractions through their semantics of dereference. The objective is to analyze the C code at the top, where the value of i ranges between 0 and 4 at the first line. This code features basic integer arithmetic and the access to an array. We focus on the evaluation of the right expression at line 3, which is written `*int (&t[i+1])` in our language (see Chapter 4). We omit the alarms: as the analyzed code does not include C undefined behaviors, there may be none with sufficiently precise abstractions.

As all the C values of the target code are constant in any memory layout, we do not need abstractions of values dependent on the layout. We thus use only some of the basic value abstractions presented in Section 5.3 for the evaluation of expressions:

- integer intervals with congruences $[i..j]r\%m$ represent sets of integer values; an interval without congruence information is written $[i..j]$, and a singleton interval is written $[i]$.

```

1  int t[5] = {1, 2, 3, 4, 5};
   if (0 <= i && i < 2)
       • r = t[i + 1];

```

	Env	Array
States D	$i \mapsto [0..1]$ $r \mapsto \perp$	$t : [1; 2; 3; 4; 5]$
$F_{*_{\text{int}}}^{\#}(D, \&i+[0])$	$[0..1]$	\top
$F_{*_{\text{int}}}^{\#}(D, \&r+[0])$	\perp	\top
$F_{*_{\text{int}}}^{\#}(D, \&t+[4..8]0\%4)$	\top	$[2..3]$
$F_{*_{\text{int}}}^{\#}(D, \&t+[0..16]0\%4)$	\top	$[1..5]$

$$\llbracket \&i \rrbracket^{\#}(D) = (\&i)^{\#} = \{ \{ \&i+[0] \} \} \quad (1)$$

$$\begin{aligned} \llbracket i \rrbracket^{\#}(D) &= \llbracket *_{\text{int}}(\&i) \rrbracket^{\#}(D) \\ &= F_{*_{\text{int}}}^{\#}(D, \{ \{ \&i+[0] \} \}) \\ &= [0..1] \sqcap_{\text{v}} \top_{\text{v}} = [0..1] \end{aligned} \quad (2)$$

$$\llbracket 1 \rrbracket^{\#}(D) = 1^{\#} = [1] \quad (3)$$

$$\begin{aligned} \llbracket i+1 \rrbracket^{\#}(D) &= \llbracket i \rrbracket^{\#}(D) +^{\#} \llbracket 1 \rrbracket^{\#}(D) \\ &= [0..1] +^{\#} [1] = [1..2] \end{aligned} \quad (4)$$

$$\llbracket \&t \rrbracket^{\#}(D) = (\&t)^{\#} = \{ \{ \&t+[0] \} \} \quad (5)$$

$$\begin{aligned} \llbracket \&t[i+1] \rrbracket^{\#}(D) &= \llbracket \&t \rrbracket^{\#}(D) +^{\#} \text{sizeof}(\text{int}) \times^{\#} \llbracket i+1 \rrbracket^{\#}(S) \\ &= \{ \{ \&t+[0] \} \} +^{\#} [4] \times^{\#} [1..2] \\ &= \{ \{ \&t+[0] \} \} +^{\#} [4..8]0\%4 \\ &= \{ \{ \&t+[4..8]0\%4 \} \} \end{aligned} \quad (6)$$

$$\begin{aligned} \llbracket *_{\text{int}}(\&t[i]) \rrbracket^{\#}(D) &= F_{*_{\text{int}}}^{\#}(D, \{ \{ \&t+[4..8]0\%4 \} \}) \\ &= \top_{\text{v}} \sqcap_{\text{v}} [2..3] = [2..3] \end{aligned} \quad (7)$$

Figure 7.1: Forward collaboration between domains

- maps $\{\{ \&x+[i..j]r\%m \}\}$ from variables to intervals-expressed byte offsets represent sets of pointer values.

We assume given two abstract domains that cooperate:

- an environment mapping integer variables to intervals representing their possible values at a statement;
- an array domain, able to represent precisely the value of each cell of an array, but also to model an imprecise access to an array;

We are not interested in their implementation or their interpretation of statements, but only in their semantics of dereferences. A representation of the state D of each domain at the bullet point in the code is given in the first line of the table. The following lines show their answers to some queries, i.e. the value abstraction computed by their dereference semantics for some location abstractions. In this example, each domain has information about different expressions, and returns \top_v for the others.

Finally, the equations of Figure 7.1 detail each step of the forward evaluation of the expression $*_{\text{int}}(\&t[i+1])$ in the state D . It proceeds bottom-up, using the domains semantics of dereference and the natural semantics of value abstractions. Note that we write the abstract value semantics $F_{\diamond}^{\#}$ with an infix notation $\diamond^{\#}$.

1. The read of the variable i is a dereference of the address $\&i$. First, the address $\&i$ is converted into its abstract representation in the value abstractions, here the map $\{\{ \&i+[0] \}\}$.
2. The abstract semantics of dereferences is provided by the domains. The environment provides the interval $[0..1]$ for the variable i . The array domain cannot interpret a non-array lvalue, and returns \top_v . The meet of those two values is $[0..1]$.
3. The constant 1 is directly converted into its abstract representation in the value abstractions, here the interval $[1]$.
4. The evaluation of the expression $i+1$ relies on the abstract semantics of intervals and congruences, leading to the abstraction $[1..2]$.
5. The constant $\&t$ is converted into the map $\{\{ \&t+[0] \}\}$.
6. The evaluation of the address $\&t[i+1]$ relies on the abstract semantics of pointer abstractions —array subscripting is equivalent to the addition of a pointer and an integer. The interval $[1..2]$ computed for the offset is multiplied by the size in bytes of the type of the array elements. Here we assumed

Listing 7.2: Assignment through a pointer

```

1  int t[5] = {1, 2, 3, 4, 5};
    int *p = t;
    if (0 <= i && i < 2)
        • r = *(p+(i+1));

```

`sizeof(int) = 4`. The resulting interval is $[4..8]$, and the congruence abstraction is able to express that the result is also congruent to 0 modulo 4. Then, this integer abstraction is added to the offset of the pointer abstraction of `&t`, which gives the new map $\{\{ \&t + [4..8]0\%4 \}\}$.

7. Finally, this dereference of the array cell is processed by the domains. The environment domain does not handle arrays, and so returns the top value abstraction. However, using the abstraction of the address computed at the previous step, the array domain is able to provide the interval $[2..3]$.

The important point in this example is the collaboration between both abstract domains, without any direct interaction. Even if the environment domain is purely numeric, it contributes to the computation of a precise abstraction for the address `&t[i+1]`. This helps the array domain to provide an accurate abstraction of the dereference, without knowing anything about the index `i+1`. Then, the environment domain will use this accurate abstraction for the assignment of `r` in the program. Here, neither domain is able to precisely interpret the statement on its own, but the cooperative evaluation allows an easy division of roles between state abstractions acting on different subsets of the C language.

In the example, both domains are numerical, and do not infer properties on pointer variables. To process dereferences or assignments through pointers, we would also need a domain inferring alias information, and able to express it by producing pointer value abstractions. For instance, listing 7.2 shows a variant of the code used in example 24, where the array `t` is accessed through the pointer `p` at line 4. Thus, interpreting this statement requires another domain providing an abstraction of the value of the pointer `p`, such as $\{\{ \&t + [0] \}\}$. As before, numerical domains may also assist in the computation of the integer abstraction of the address offset.

7.1.2 Additional Query on any Expressions

The abstract semantics of dereferences is the only mandatory feature required from an abstract domain to achieve a complete evaluation of any expression. Indeed, dereferences depend on memories —and

thus on concrete states, and must be approximated by state abstractions. All other operations on expressions can be safely handled by value (and location) abstractions. Yet, an abstract domain may infer relevant properties about any expressions, and not only dereferences. For instance, a domain tracking inequalities in a program may express that $e_1 - e_2$ is positive when it has inferred $e_1 \geq e_2$, regardless of the value of e_1 and e_2 . To take advantage of all inferences made by the domains, the evaluation also involves the abstract states on arbitrary expressions.

The second query provided by an abstract domain supplies additional information about arbitrary C expressions. It computes sound abstractions for their complete evaluations in all the concrete states in the concretization of the abstract state.

Definition 52 (Arbitrary domain query). In an abstract state D , a sound query $F_{\mathbb{D}}^{\#}(D, e)$ for an expression e computes:

- a sound value abstraction v of the concrete values that the expression e may have in a concrete state of $\gamma_{\mathbb{D}}(D)$;
- an alarm map \mathbf{A} which is a sound abstraction of the evaluation of e in any concrete state of $\gamma_{\mathbb{D}}(D)$.

$$F_{\mathbb{D}}^{\#} : \mathbb{D} \rightarrow \text{expr} \rightarrow \mathbb{V}^{\#} \times \mathbb{A}$$

$$\forall S \in \gamma_{\mathbb{D}}(D), F_{\mathbb{D}}^{\#}(D, e) \models_{\mathbb{V} \times \mathbb{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S)$$

Proposition 5. For a product of domains $\mathbb{D} = \mathbb{D}_1 \times \dots \times \mathbb{D}_n$, a sound abstract semantics $F_{\mathbb{D}}^{\#}$ can be defined by:

$$F_{\mathbb{D}}^{\#}((d_1, d_2, \dots, d_n), l) = F_{\mathbb{D}}^{\#}(d_1, l) \sqcap_{\mathbb{V} \times \mathbb{A}} F_{\mathbb{D}}^{\#}(d_2, l) \sqcap_{\mathbb{V} \times \mathbb{A}} \dots \sqcap_{\mathbb{V} \times \mathbb{A}} F_{\mathbb{D}}^{\#}(d_n, l)$$

Proof. The same reasoning as for proposition 4 applies here. \square

In [EVA](#), the evaluation of an expression queries the state domains on each non-constant expression, using $F_{*_{\tau}}^{\#}$ for dereferences and $F_{\mathbb{D}}^{\#}$ for other expressions. In the second case, the meet $\sqcap_{\mathbb{V} \times \mathbb{A}}$ is used to intersect the abstractions computed by the abstract semantics of values and by the query of domains.

Example 25. The analysis of the code shown at [Figure 7.2](#) illustrates the utility of the query on arbitrary expressions. We use integer intervals as value abstractions, and two state abstractions: an environment mapping variables to intervals, and a relational domain inferring symbolic inequalities between C expressions. The two assertions constrain the possible values of the variables a and b between 0 and 10, which are stored as intervals in the environment domain. The condition at line 3 does not allow reducing these intervals, but the symbolic domain retains that the inequality $a+b > 9$ holds. The table shows the abstract states inferred at line 4, and their answers to queries on some expressions, including but not limited to dereferences. The evaluation of the expression $2*(a+b)$ is as follow:

1

```
/*@ assert 0 <= a <= 10; */
/*@ assert 0 <= b <= 10; */
if (a + b >= 10)
  • r = 2 * (a + b);
```

	Environment	Inequalities
States D	$a \mapsto [0..10]$ $b \mapsto [0..10]$ $r \mapsto \perp$	$a + b > 9$
a	$[0..10]$	\top
b	$[0..10]$	\top
$a + b$	\top	$[10..+\infty]$
$2 * (a + b)$	\top	\top

Figure 7.2: Collaboration between domains on arbitrary expressions

- the environment domain provides the interval $[0..10]$ for the variables a and b .
- the environment domain provides no value abstraction for the expression $a+b$, but the interval value semantics of addition computes the interval $[0..20]$. The relational domain supplies the interval $[10..+\infty]$. The meet of both intervals is $[10..20]$.
- the interval semantics of multiplication computes $[20..40]$ for the expression $2*(a+b)$, and no domain can reduce this value.

Here, the relational domain cannot avail its inferred inequality without the query on arbitrary expressions, which would have led to a significant loss of precision when interpreting the assignment.

7.1.3 Interaction through an Oracle

To compute a precise value abstraction for an expression, a domain—and especially a relational one—may need additional information about other expressions. Thus, a domain can request the evaluation of new expressions, through an oracle given in argument of the query functions. The oracle triggers the requested evaluation using all available domains, and returns the cooperatively computed abstractions to the initial domain. The oracle has the same specification as the forward evaluation function: it provides an alarm map and a value that are sound approximations of the concrete evaluation of the expression in the current state D . A domain can thus rely on it to make its own answer to a query.

Definition 53 (Oracle). An oracle is a function from expressions to pairs of a value abstraction and an alarm map.

$$\text{oracle} : \text{expr} \rightarrow \mathbb{V}^\# \times \mathbb{A}$$

Definition 54 (Oracle concretization). For an oracle oracle , we define $\gamma_o(\text{oracle})$ as the set of concrete states S for which the answer of the oracle for any expression e is a sound approximation of its evaluation in S :

$$\gamma_o(\text{oracle}) \triangleq \{S \in \mathcal{S} \mid \forall e \in \text{expr}, \text{oracle}(e) \models_{\mathbb{V} \times \mathbb{A}} \overline{\llbracket e \rrbracket}^\Theta(S)\}$$

Definition 55 (Sound oracle). In an abstract state D and a valuation \mathcal{E} , an oracle is *sound* if it produces sound approximations of the evaluations in the concrete state of $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$.

$$\forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \forall e \in \text{expr}, \text{oracle}(e) \models_{\mathbb{V} \times \mathbb{A}} \overline{\llbracket e \rrbracket}^\Theta(S)$$

This is equivalent to $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}) \subseteq \gamma_o(\text{oracle})$.

Definition 56. The domain queries $F_{*\tau}^\#$ and $F_{\mathbb{D}}^\#$ of an abstract state D take as additional argument an oracle. The queries produce sound alarm maps and value abstractions for the concrete states in $\gamma_{\mathbb{D}}(D) \cap \gamma_o(\text{oracle})$.

$$\begin{aligned} F_{*\tau}^\# &: (\text{expr} \rightarrow \mathbb{V}^\# \times \mathbb{A}) \rightarrow \mathbb{D} \rightarrow \text{expr} \rightarrow \mathbb{L}^\# \rightarrow \mathbb{V}^\# \times \mathbb{A} \\ F_{\mathbb{D}}^\# &: (\text{expr} \rightarrow \mathbb{V}^\# \times \mathbb{A}) \rightarrow \mathbb{D} \rightarrow \text{expr} \rightarrow \mathbb{V}^\# \times \mathbb{A} \end{aligned}$$

$$\forall D \in \mathbb{D}, \forall e \in \text{expr}, \forall l \in \mathbb{L}, \forall a \in \gamma_v(l), \forall S \in \gamma_{\mathbb{D}}(D) \cap \gamma_o(\text{oracle}),$$

$$\begin{cases} F_{*\tau}^\#(\text{oracle}, D, *_{\tau}a, l) \models_{\mathbb{V} \times \mathbb{A}} \overline{\llbracket *_{\tau}a \rrbracket}^\Theta(S) \\ F_{\mathbb{D}}^\#(\text{oracle}, D, e) \models_{\mathbb{V} \times \mathbb{A}} \overline{\llbracket e \rrbracket}^\Theta(S) \end{cases}$$

Theorem 9. If the oracle given to a domain is sound in the current abstract state and valuation, then the requirement of the queries of definition 56 still guarantees the soundness of the forward evaluation stated by proposition 1 and proved by theorem 6.

Proof. According to proposition 1, a sound forward evaluation in an abstract state D and a valuation \mathcal{E} produces abstractions (a new valuation and an alarm map) that are over-approximations of the concrete evaluation in any concrete states in $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$. By definition 55, if an oracle is sound for D and \mathcal{E} , then:

$$\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}) \subseteq \gamma_o(\text{oracle}) \quad (7.1)$$

We can informally see that refining the query of a state D to an approximation of the concrete states in $\gamma_{\mathbb{D}}(D) \cap \gamma_o(\text{oracle})$ instead of just $\gamma_{\mathbb{D}}(D)$ does not impact the soundness of the evaluation.

Listing 7.3: Implementation of an oracle in the evaluator

```

1 let rec forward_eval fuel state expr =
  try Valuation.find !valuation expr
  with Not_found ->
    Valuation.add !valuation expr (Value.top, Alarmset.all);
5   let r = coop_forward_eval fuel state expr in
    Valuation.add !valuation expr r; r

and coop_forward_eval fuel state expr =
  let oracle =
10   if fuel > 0
    then forward_eval (pred fuel) state
    else (Value.top, Alarmset.all)
  in
15  let v, a = internal_forward_eval fuel state expr in
    let v', a' = Domain.extract_expr oracle state expr in
    Value.meet v v', Alarms.meet a a'

and internal_forward_eval fuel state expr =
  [...]

```

In order to formally verify this fact, we would need to redo the proof of theorem 6. This proof relies on the invariant of the atomic updates of the valuation, stated by theorem 5, which itself relies on lemma 22. This lemma asserts that the forward semantics used for an atomic update produces sound value abstractions of the concrete evaluation of the expression being processed. In other words, for an expression e , the forward abstract semantics must compute a value v such that:

$$\forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(v) \cup \{\Omega\}$$

By inclusion 7.1, a concrete state in $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$ is also in $\gamma_o(\text{oracle})$, and thus the queries of the domain produce value abstractions that satisfy this equation. The same reasoning applies for the alarm maps. \square

A forward evaluation satisfying the properties of propositions 1 and 3 is a sound oracle. As the forward evaluation returns a valuation, the oracle just needs to extract the expected value abstraction from the valuation.

Proposition 6. *In an abstract state D and a valuation \mathcal{E} , a sound oracle can be defined by:*

$$\text{oracle}_{D, \mathcal{E}}(e) \triangleq (\mathcal{E}'(e), \mathbf{A}) \quad \text{where } (\mathcal{E}', \mathbf{A}) = \llbracket e \rrbracket_D^{\#}(\mathcal{E})$$

Proof. Let S be a concrete state in $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$. Proposition 1 ensures that:

$$\begin{aligned} \mathbf{A} &\models_{\mathbf{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S) \\ \overline{\llbracket e \rrbracket}^{\Theta}(S) \neq \Omega &\Rightarrow S \in \gamma_{\mathbb{E}}(\mathcal{E}') \end{aligned}$$

By definition 46, $S \in \gamma_{\mathbb{E}}(\mathcal{E}') \Rightarrow \overline{\llbracket e \rrbracket}^{\Theta}(S) \in \gamma_v(\mathcal{E}'(e))$ —proposition 3 ensures that $e \in \text{dom}(\mathcal{E}')$. We thus have $\overline{\llbracket e \rrbracket}^{\Theta}(S) \in \gamma_v(\mathcal{E}'(e)) \cup \{\Omega\}$ in all cases, and then:

$$(\mathcal{E}'(e), \mathbf{A}) \models_{\mathbb{V} \times \mathbf{A}} \overline{\llbracket e \rrbracket}^{\Theta}(S)$$

□

As the oracle triggers new evaluations, an uncontrolled use of the oracle may lead to (1) a loop in the evaluation, or to (2) an infinite chain of evaluations of different expressions. To prevent (1) from happening, the oracle can return \top on re-occurrences of the same expression. The number of recursive uses of the oracle should also be limited by a parameter of the analysis, to avoid (2).

Listing 7.3 contains an improvement of the simplified evaluator presented in Section 6.1.7, that shows the implementation of the oracle used in EVA. The `extract_expr` query of the domain takes as argument the oracle, which calls the main forward evaluation. The evaluation uses a valuation as a cache. Before any computation on an expression e , it also binds the top value and alarm abstractions to e in the valuation. This prevents any loop in the evaluation: if a domain requests the re-evaluation of e , the main function returns directly this top abstractions. This imprecise value is replaced by the computed one at the end of the evaluation. Moreover, an integer fuel is used to avoid an infinite sequence of evaluations: a call of the oracle decrements it, and the oracle returns the top abstractions when the fuel reaches zero.

Thanks to the oracle, the abstract domains may share information through alarms and value abstractions during the evaluation, without resorting to a direct communication between domains. Especially, the oracle allows a relational domain to fully avail the relations it has inferred, and lets the other domains collaborate in leveraging these relations. The following example illustrates this with a symbolic equality domain.

Example 26. In this example, we study the analysis of the code snippet shown in figure 7.3, which is quite similar to the code of example 24: at line 5, the assignment of the variable r depends on the value of $t[i]$, under the condition $i < 2$. However, the access to the array cell $t[i]$ is done through the `tmp` variable, which has been computed before the condition. Accurately interpreting the assignment

thus requires to take into account the reduction of tmp implied by the condition $i < 2$.

We use the same value abstractions as in the previous example: intervals and congruences for integers, and maps from variables to integer abstractions for pointers. We assume given three abstract domains —the first two being those already used in the example 24. Their abstract states before the interpretation of the assignment of line 5 is given in the first line of the figure table. The following lines show their answers to some queries about various expressions. Once again, the domains supply the top value abstraction on expressions that they do not handle, even if they have inferred relevant properties about the subterms: they let the interval semantics compute a precise value for these expressions from the values of the subterms. The domains used in this example are:

- an environment mapping integer variables to intervals. After the initial assertion $0 \leq i$ and the condition $i < 2$, the variable i is known to be in $[0..1]$. The set of possible values of tmp has been cooperatively computed at line 3, but has not been reduced by the condition, since this domain is unable to link i and tmp .
- the array domain that represents the values of each cell of t
- a relational domain inferring symbolic equalities between expressions. At line 3, it has inferred $tmp = t[i] + 1$, which is still valid at line 5.

Without the oracle mechanism, the evaluation of $2 * tmp$ in these states leads simply to the abstraction $[2] *^\# [2..6] = [4..12]0\%2$. The equality domain cannot supply a value abstraction for tmp by itself, despite the relation it has inferred. But it can use the oracle to get a value abstraction of $t[i] + 1$, and then returns it as a sound value abstraction of tmp . The oracle calls the complete evaluation of $t[i] + 1$ in the full state, which proceeds as shown previously in example 24: by intertwining the value semantics and the semantics of dereference provided by the environment and the array domain, it computes the precise interval $[2..3]$ for $t[i] + 1$. Indeed, the standard evaluation of $t[i] + 1$ takes into account the reduction of i stemming from the condition at line 4. Then, the equality domain returns the interval $[2..3]$ for tmp , and the evaluation of $2 * tmp$ becomes $[2] *^\# [2..3] = [4..6]0\%2$.

Note that when evaluating $t[i] + 1$, the equality domain can request the evaluation of tmp through the oracle, for the same reason as before. Here, the oracle returns the top abstraction directly, to prevent the evaluation to loop endlessly between tmp and $t[i] + 1$.

```

1  /*@ assert 0 <= i <= 4; */
   int t[5] = {1, 2, 3, 4, 5};
   int tmp = t[i] + 1;
   if (i < 2)
5     • r = 2 * tmp;

```

	Env	Array	Equality
States D	$i \mapsto [0..1]$ $tmp \mapsto [2..6]$ $r \mapsto \perp$	$t :$ $[1; 2; 3; 4; 5]$	$tmp = t[i] + 1$
$2 * tmp$	\top	\top	\top
tmp	$[2..6]$	\top	$\text{oracle}_D(t[i] + 1)$
i	$[0..1]$	\top	\top
$t[i]$	\top	$[1..2]$	\top
$t[i] + 1$	\top	\top	$\text{oracle}_D(tmp)$
r	\perp	\top	\top

Figure 7.3: Interaction through the oracle

Listing 7.4: Signature of domain backward propagators

```

1  type 'a or_bottom = 'Bottom | 'Value of 'a
   val backward_location : (* B*τ# *)
4     state -> location -> value -> location or_bottom
5
   val reduce_further : (* BD# *)
7     state -> exp -> value -> (exp * value) list

```

7.2 BACKWARD PROPAGATORS

Within value abstractions, forward and backward propagators are dual. Likewise, state abstractions can provide the backward counterparts of queries. Listing 7.4 presents their signature.

7.2.1 Backward Semantics of Dereference

When an abstraction of the possible values of an lvalue is reduced, the abstraction of the memory location of the lvalue might be reduced as well. This is typically the case for memory accesses through an imprecise pointer or in an array. This reduction requires knowledge of the memory state, and thus involves the state abstractions.

Definition 57 (Backward semantics of dereference). In an abstract state D , a sound backward semantics $B_{*_{\tau}}^{\#}$ of dereference is a function from a location abstraction l and a value abstraction v to a new location abstraction l' such that l' over-approximates all concrete addresses in $\gamma_v(l)$ whose dereference is in $\gamma_v(v)$ for at least one concrete state of $\gamma_D(D)$.

$$B_{*_{\tau}}^{\#} : \mathbb{D} \rightarrow \mathbb{L}^{\#} \rightarrow \mathbb{V}^{\#} \rightarrow \mathbb{L}^{\#}$$

$$\{a \in \gamma_L(l) \mid \exists S \in \gamma_D(D), \overline{[*_{\tau}a]}^{\Theta}(S) \in \gamma_v(v)\} \subseteq \gamma_L(B_{*_{\tau}}^{\#}(D, l, v))$$

As usual for backward propagators, it is always sound to return the previous location unreduced.

Proposition 7. For a product of domains $\mathbb{D} = \mathbb{D}_1 \times \dots \times \mathbb{D}_n$, a sound abstract semantics $B_{*_{\tau}}^{\#}$ can be defined by:

$$B_{*_{\tau}}^{\#}((d_1, \dots, d_n), l, v) \triangleq B_{*_{\tau}}^{\#}(d_1, l, v) \sqcap_{\mathbb{V}} \dots \sqcap_{\mathbb{V}} B_{*_{\tau}}^{\#}(d_n, l, v)$$

Proof. Let $D = (d_1, \dots, d_n)$ be a product of abstract states of a domain $\mathbb{D} = \mathbb{D}_1 \times \dots \times \mathbb{D}_n$. Let l be a location abstraction and v be a value abstraction. Let a be a constant pointer value such that:

$$a \in \gamma_L(l) \wedge \overline{[*_{\tau}a]}^{\Theta}(S) \in \gamma_v(v)$$

We need to prove that $a \in \gamma_L(B_{*_{\tau}}^{\#}(D, l, v))$. Let S be a state in $\gamma_D(D)$ such that $\overline{[*_{\tau}a]}^{\Theta}(S) \in \gamma_v(v)$. For each i between 1 and n , we have $\gamma_D(D) \subseteq \gamma_D(d_i)$ and thus $S \in \gamma_D(d_i)$. This implies:

$$a \in \gamma_L(l) \wedge \forall i \in \{1, \dots, n\}, \exists S \in \gamma_D(d_i), \overline{[*_{\tau}a]}^{\Theta}(S) \in \gamma_v(v)$$

By definition 57:

$$\forall i \in \{1, \dots, n\}, a \in \gamma_L(B_{*_{\tau}}^{\#}(d_i, l, v))$$

By soundness of the meet of value abstractions, we can deduce:

$$a \in B_{*_{\tau}}^{\#}(d_1, l, v) \sqcap_{\mathbb{V}} \dots \sqcap_{\mathbb{V}} B_{*_{\tau}}^{\#}(d_n, l, v) = \gamma_L(B_{*_{\tau}}^{\#}(D, l, v))$$

□

Example 27. The analysis of the code snippet of Figure 7.4 illustrates the need for a backward propagator for dereferences. We use the environment domain and the array domain already used in the previous examples. Their abstract states at line 3 are represented in the figure table. In these states, a forward evaluation of the condition of line 4 computes:

- the interval $[1..4]$ for the expression $i+1$, according to the abstract semantics of addition.

1	<code>/*@ assert 0 <= i <= 3; */</code>
	<code>int t[5] = {1, 2, 3, 4, 5};</code>
	•
4	<code>if (t[i+1] < 4)</code>
5	<code> r = 2 * i;</code>

	Env	Array
States D	$i \mapsto [0..3]$ $r \mapsto \perp$	$t : [1; 2; 3; 4; 5]$

Figure 7.4: Backward propagation on an array access

- the map $\{\{ \&t+[4..16]0\%4 \}\}$ for the address $\&t[i+1]$, according to the abstract semantics of array subscripting.
- the interval $[2..5]$ for the expression $t[i+1]$, through the dereference semantics of the array domain.
- the interval $[0..1]$ for the condition $t[i+1] < 4$, according to the abstract semantics of comparison.

Then, the analysis of the *if* branch assumes the condition to be true by backward propagating the singleton interval $[1]$ through the expression $t[i+1] < 4$. This backward evaluation reduces:

- the interval for $t[i+1]$ to $[2..3]$, according to the backward propagator of intervals on comparisons.
- the map for the address $\&t[i+1]$ to $\{\{ \&t+[4..8]0\%4 \}\}$, thanks to the backward propagation provided by the array domain. Indeed, this map represents the only locations of the previous abstraction of the address for which the dereference has a value in $[2..3]$.
- the interval for the expression $i+1$ to $[1..2]$, through the backward semantics of array subscripting.
- the interval for the variable i to $[0..1]$, according to the backward semantics of addition.

The backward semantics of dereference allows reducing the value abstraction of $i+1$ according to the value abstraction $t[i+1]$. If the variable i is used later in the branch, this reduction can be significant for the precision of the analysis.

Example 28. The previous example exhibits a reduction of an array index according to the value of the array cell. The code of Figure 7.5 presents a reduction of a pointer according to the pointed value. We use here the environment domain for integer variables, along with

<pre> 1 int a = 42; int b = 0; int v = rand(); int *p = v ? &a : &b; 5 • 6 int r = 100 / *p; </pre>		
	Environment	Alias analysis
States D	$a \mapsto [42] \quad v \mapsto \top$	$p \rightarrow \{ \{ \&a+[0], \&b+[0] \} \}$
	$b \mapsto [0] \quad r \mapsto \perp$	

Figure 7.5: Backward propagation on a pointer dereference

a domain performing an alias analysis and inferring pointer abstractions (maps from variables to integer offsets). The table presents their states at line 5. When evaluating the expression at line 6, the aliasing domain provides the abstraction $\{ \{ \&a+[0], \&b+[0] \} \}$ for the pointer p . Using it, the environment domain abstracts the result of the dereference $*p$ into the interval $[0..42]$, by joining the intervals it stores at each location $\{ \{ \&a+[0] \} \}$ and $\{ \{ \&b+[0] \} \}$. Then, the forward evaluation of $100 / *p$ raises an alarm about the potential division by zero.

We have seen in Section 6.2.1 that an alarm makes a backward propagation relevant. Here, the interval backward semantics on division reduces the abstraction of the divisor $*p$ to $[1..42]$. Then, the environment domain can reduce the abstraction of the dereference location: as b is equal to 0, $\{ \{ \&a+[0] \} \}$ is the only possible location for this dereference. This reduces the abstraction of p for the alias analysis.

7.2.2 Triggering New Reductions

On a backward propagation, the relations inferred by a domain may induce further interesting reductions. For instance, if $a \leq b$ holds, then any reduction of the infimum of the possible values of a implies the same reduction for b . Hence, when performing a reduction, the generic evaluator notifies the domains through the function $B_{\mathbb{D}}^{\#}$, which returns a set of new reductions to be backward propagated by the evaluator. The new reductions, deduced from the prior one and from the inferences made by the domain, must be correct in the concrete states for which the initial reductions are valid. To avoid diverging, the generic evaluator limits the number of successive reductions made through $B_{\mathbb{D}}^{\#}$. In the signature of Listing 7.4, $B_{\mathbb{D}}^{\#}$ is named `reduce_further`, and we call it the *reducer*.

Definition 58. In an abstract state D , a reduction of an expression e to a value abstraction v soundly implies a reduction of an expression e'

to a value abstraction v' when for all concrete state S in $\gamma_{\mathbb{D}}(D)$ in which the evaluation of e is in $\gamma_v(v)$, the evaluation of e' is in $\gamma_v(v')$.

The query $B_{\mathbb{D}}^{\#}(D, e, v)$ returns a set of pairs (e', v') such that the reduction of e to v soundly implies the reduction of e' to v' .

$$\begin{aligned} B_{\mathbb{D}}^{\#} : \mathbb{D} \rightarrow \text{expr} \rightarrow \mathbb{V}^{\#} \rightarrow \mathcal{P}(\text{expr} \times \mathbb{V}^{\#}) \\ \forall (e', v') \in B_{\mathbb{D}}^{\#}(D, e, v), \forall S \in \gamma_{\mathbb{D}}(D), \\ \llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(v) \Rightarrow \llbracket e' \rrbracket^{\Theta}(S) \in \gamma_v(v') \end{aligned}$$

Proposition 8. *For a product of domains $\mathbb{D} = \mathbb{D}_1 \times \dots \times \mathbb{D}_n$, a sound abstract semantics $B_{\mathbb{D}}^{\#}$ can be defined as the union of all the new reductions proposed by each domain \mathbb{D}_i .*

Proof. Let $D = (d_1, \dots, d_n)$ be a product of abstract states of domains $\mathbb{D}_1, \dots, \mathbb{D}_n$. Let e and e' be two expressions, and v and v' two value abstractions, and let $i \in \{1, \dots, n\}$. If (e', v') is a new reduction proposed by the domain \mathbb{D}_i as a sound consequence of the reduction of e to v , then:

$$\forall S \in \gamma_{\mathbb{D}}(d_i), \llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(v) \Rightarrow \llbracket e' \rrbracket^{\Theta}(S) \in \gamma_v(v')$$

As $\gamma_{\mathbb{D}}(D) \subseteq \gamma_{\mathbb{D}}(d_i)$, this implies:

$$\forall S \in \gamma_{\mathbb{D}}(D), \llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(v) \Rightarrow \llbracket e' \rrbracket^{\Theta}(S) \in \gamma_v(v')$$

and thus (e', v') is also a valid new reduction in the state D . \square

Theorem 10. *The backward propagations of the reductions proposed by a domain through $B_{\mathbb{D}}^{\#}(D, e, v)$ do not impact the soundness of the backward evaluation stated by proposition 2 and proved by theorem 6.*

Proof. Let D be an abstract state, \mathcal{E} be a valuation, e be an expression and v be a value abstraction. Proposition 2 states that a sound backward propagation $\llbracket e \rrbracket_D^{\#}(\mathcal{E}, v)$ produces a valuation \mathcal{E}' such that:

$$\forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(v) \Rightarrow S \in \gamma_{\mathbb{E}}(\mathcal{E}')$$

We consider a new reduction proposed by the domain:

$$(e', v') \in B_{\mathbb{D}}^{\#}(D, e, v)$$

Then we have:

$$\forall S \in \gamma_{\mathbb{D}}(D), \llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(v) \Rightarrow \llbracket e' \rrbracket^{\Theta}(S) \in \gamma_v(v')$$

As $\gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}) \subseteq \gamma_{\mathbb{D}}(D)$, we also have:

$$\forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \llbracket e \rrbracket^{\Theta}(S) \in \gamma_v(v) \Rightarrow \llbracket e' \rrbracket^{\Theta}(S) \in \gamma_v(v') \quad (7.2)$$

<pre> 1 /*@ assert 0 <= i <= 4; */ int t[] = {1, 2, 3, 4, 5}; int tmp = t[i] + 1; • 5 if (tmp < 4) [...] </pre>			
	Environment	Array domain	Equalities
States D	$i \mapsto [0..4]$ $tmp \mapsto [2..6]$	$t : [1; 2; 3; 4; 5]$	$tmp = t[i] + 1$

Figure 7.6: Triggering new reductions

By proposition 2 again, we know that the backward propagation $\llbracket e' \rrbracket_D^\#(\mathcal{E}', v')$ produces a valuation \mathcal{E}'' that satisfies:

$$\forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \llbracket e' \rrbracket^\Theta(S) \in \gamma_v(v') \Rightarrow S \in \gamma_{\mathbb{E}}(\mathcal{E}'') \quad (7.3)$$

By 7.2 and 7.3, we have:

$$\forall S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), \llbracket e \rrbracket^\Theta(S) \in \gamma_v(v) \Rightarrow S \in \gamma_{\mathbb{E}}(\mathcal{E}'')$$

We can conclude that the sequence of two backward propagations $\llbracket e' \rrbracket_D^\#(\llbracket e \rrbracket_D^\#(\mathcal{E}, v), v')$ satisfies the property of proposition 2. \square

In a way, the reducer $B_{\mathbb{D}}^\#$ is the backward counterpart of the oracle: the oracle allows a domain to start new forward evaluations, while the reducer allows it to start new backward propagations. As for the oracle, letting a domain start new backward propagations may lead to a loop in the evaluation, or to an infinite chain of propagations on different expressions. The same solutions can be applied to solve this issue: the number of recursive uses of $B_{\mathbb{D}}^\#$ should be limited, using a decreasing fuel, and re-occurrences of the same expression should be detected.

Example 29. We point out the interest for a domain to propagate new reductions on the analysis of the *if* statement at line 5 of the code of Figure 7.6. We use the environment, array and equality domain shown previously. The table shows their states at the bullet point of line 4. On the *if* branch, the backward propagation of the value [1] for the condition reduces the abstraction for *tmp* to the interval [2..3]. The evaluation notifies the domains of this reduction. As the equality domain has inferred $tmp = t[i] + 1$, it can start the backward propagation of the interval [2..3] to the other expression $t[i] + 1$. This will lead to the reduction of the variable *i* to the interval [0..1], allowing further precision when analyzing the branch.

Both the oracle and the reducer allow a domain to avail its inferred relation during an evaluation. It may not always be obvious for a

<pre> 1 /*@ assert 0 <= x <= 20; */ int y = x; if (y < 10) r = x; </pre>	<pre> 1 /*@ assert 0 <= x <= 20; */ int y = x; if (y < 10) { y = 0; r = x; 5 } </pre>
--	--

Figure 7.7: Using the oracle or the reducer

domain to determine when to use either of the functionalities. On the left code of Figure 7.7, an equality domain can use the reducer at line 3 to propagate that not only y but also x is smaller than 10. If it does not, it can also use the oracle at line 4 to evaluate y , and return the result for x . In both cases, the equality domain assists in interpreting the assignment, leading the abstraction of r to reflect that the variable is smaller than 10. Using the reducer narrows the abstraction of x as soon as possible, even if it is not read in the branch. On the other hand, relying on the oracle postpones the reduction of the abstraction of x to when it becomes useful (i.e. when x is read). However, if y is written in the branch before the assignment of r , then the equality does not hold anymore, and x cannot be reduced according to the condition $y < 10$. On the right code of Figure 7.7, using the reducer when backward propagating the condition is the only way to allow a precise interpretation of the assignment at line 5.

On some code patterns, only one of the oracle or reducer can be of some help. On the code of Figure 7.6, the equality $tmp = t[i] + 1$ allows a reduction of the abstraction of i through the reducer. But the equality domain has inferred no equality about i itself, and thus cannot use the oracle to narrow the abstraction of i . With the oracle, the domain can only narrow the abstraction of $t[i] + 1$ in the branch. It is possible to deduce the value of i from the value of $t[i] + 1$ by a backward propagation, but the oracle does not provide this feature. On the other hand, on the code of Figure 7.3, the same equality $tmp = t[i] + 1$ does not allow reducing the abstraction of tmp from the reduction of i . This would need a forward evaluation of $t[i] + 1$ first, and the reducer does not provide this feature.

This restriction of the oracle and the reducer—the oracle does only forward evaluation, and the reducer only backward propagation—helps limiting their cost. It also makes their implementation much easier, by keeping separate the forward and the backward propagations.

7.3 ABSTRACT SEMANTICS OF STATEMENTS

An abstract domains implements an abstract semantics of statements through abstract transformers (or transfer functions). Such transform-

ers over-approximate the effects of a statement on an element of the domain. These transformers are used to propagate abstract states through the CFG, inferring properties about the possible behaviors that may occur at a program point during its concrete execution.

Until now, we have focused on the cooperative evaluation of expressions to alarms and value abstractions. We will now see how this evaluation assists the domain transformers to accurately interpret a statement, using information provided by all abstract domains.

7.3.1 Abstract Semantics of Statements

When interpreting a statement, *EVA* starts by evaluating cooperatively all involved addresses and expressions into value and location abstractions. It follows the strategy described in Chapter 6, and involves the abstract domains as explained above. The evaluations produce an alarm map that over-approximates the undesirable behaviors that may occur at this statement, according to the current abstract state. Using the internal features of the FRAMA-C kernel, *EVA* emits the unproven alarms (with a non-true status) as ACSL assertions that report the potential undesirable behaviors to the final user.

Then, *EVA* interprets the statement semantics in the case where no undesirable behavior happens: it considers only the concrete states in which the concrete evaluations succeed, which are the concrete states that satisfy the emitted assertions. The valuation produced by the evaluations is a sound abstraction of these concrete states: this is guaranteed by the invariant of the atomic updates of a valuation, proved in theorem 5. The abstract transformers of a domain can safely use this valuation to model the effect of the statement on their abstract states.

Definition 59 (Domain transformers). A domain transformer is a function from a statement, a valuation and an abstract state to a new abstract state that is a sound over-approximation of the non-erroneous concrete states after the statement.

$$\begin{aligned}
 T_{\mathbb{D}} : \text{stmt} &\rightarrow \mathbb{E} \rightarrow \mathbb{D} \rightarrow \mathbb{D} \\
 \forall \text{stmt} \in \text{stmt}, \forall \mathcal{E} \in \mathbb{E}, \forall D \in \mathbb{D}, \\
 \gamma_{\mathbb{D}}(T_{\mathbb{D}}(\text{stmt}, \mathcal{E}, D)) &\supseteq \{S' \in \mathcal{S} \setminus \{\Omega\} \mid \\
 &\quad \exists S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}), S' = \overline{\{\text{stmt}\}}(S)\}
 \end{aligned}$$

The content of the valuation \mathcal{E} given to a domain transformer depends on the interpreted statement. The valuation is the result of:

- for an assignment $*_{\tau}a := e$, the evaluation of the expression e to a value abstraction and of the address a to a location abstraction;
- for a copy $*_{\tau}a \leftarrow *_{\tau}a'$, the evaluation of both addresses a and a' ;

- for an assumption $e == 0?$, the evaluation of the expression e and the backward propagation of the value abstraction $0^\#$ for this expression;
- for the entry in scope or exit from scope of variables, the valuation is empty as the statement does not involve any expression or address. Actually, the domain transformers for these statements take no valuation.

The soundness properties of the evaluation functions have been stated in Chapter 6. They ensure that the use of the domain transformers described by definition 59 on these valuations leads to sound transfer functions over-approximating the concrete semantics of statements. This is formally proved by the following theorems for assignments, copies and assumptions. Since the entry in scope and exit from scope cannot fail and do not involve valuations, their abstract transformers fulfill directly the requirement of definition 8.

Theorem 11. *Let stmt be the assignment $*_\tau a := e$, and let D be an abstract state of the domain \mathbb{D} . If $\llbracket e \rrbracket_D^\# : \mathbb{E} \rightarrow \mathbb{E} \times \mathbb{A}$ is an evaluation function satisfying the properties of proposition 1, then the function $\{\text{stmt}\}^\#$ defined as follows is a sound transfer function on non erroneous states, according to definition 8.*

$$\{\text{stmt}\}^\#(D) = T_{\mathbb{D}}(\text{stmt}, \mathcal{E}, D) \quad \text{where } \mathcal{E}, \mathbf{A} = \llbracket a \rrbracket_D^\#(\mathcal{E}') \\ \text{and } \mathcal{E}', \mathbf{A}' = \llbracket e \rrbracket_D^\#(\top_{\mathbb{E}})$$

Moreover, if it exists a concrete state S in $\gamma_{\mathbb{D}}(D)$ for which the execution of stmt fails (i.e. such that $\{\text{stmt}\}(S) = \Omega$), then \mathbf{A} or \mathbf{A}' is a non-bottom alarm map, and an alarm is thus emitted by the analyzer (EVA emits all alarms of $\mathbf{A} \sqcup_{\mathbb{A}} \mathbf{A}'$ that have a non-true status).

Proof. Let S be a concrete state of $\gamma_{\mathbb{D}}(D)$. The semantics of assignment, given in Figure 4.9, defines $\{\text{stmt}\}(S) = \Omega$ only if $\llbracket e \rrbracket^\Theta(S) = \Omega$ or $\llbracket a \rrbracket^\Theta(S) = \Omega$. Otherwise, the execution of stmt in S succeeds.

- Case 1.* $\llbracket e \rrbracket^\Theta(S) = \Omega$. By lemma 5, $\exists A \in \text{alarms}(e)$, $\neg A(S)$. As $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \top_{\mathbb{E}})$, the soundness of the evaluation (proposition 1) ensures that $\mathbf{A}' \models_{\mathbb{A}} \llbracket e \rrbracket^\Theta(S)$. By definition 38, $\mathbf{A}(A) \neq \text{true}$. Therefore, \mathbf{A} is not the bottom alarm map, and the alarm A is emitted by the analyzer.
- Case 2.* $\llbracket e \rrbracket^\Theta(S) \neq \Omega$ but $\llbracket a \rrbracket^\Theta(S) = \Omega$. As $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \top_{\mathbb{E}})$ and $\llbracket e \rrbracket^\Theta(S) \neq \Omega$, proposition 1 ensures that $S \in \gamma_{\mathbb{E}}(\mathcal{E}')$. Thus, $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}')$ and the same reasoning as before applies: $\exists A \in \text{alarms}(a)$, $\neg A(S)$, and $\mathbf{A} \models_{\mathbb{A}} \llbracket e \rrbracket^\Theta(S)$, and finally $\mathbf{A}(A) \neq \text{true}$. The alarm A is emitted by the analyzer.

Case 3. $\llbracket e \rrbracket^\Theta(S) \neq \Omega$ and $\llbracket a \rrbracket^\Theta(S) \neq \Omega$.

By proposition 1, we have $S \in \gamma_{\mathbb{E}}(\mathcal{E}')$.

Thus $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E}')$, and we also have $S \in \gamma_{\mathbb{E}}(\mathcal{E})$.

Finally, $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$ and definition 59 ensures that:

$$\llbracket \text{stmt} \rrbracket(S) \in \gamma_{\mathbb{D}}(\mathbf{T}_{\mathbb{D}}(\text{stmt}, \mathcal{E}, D))$$

Finally, $\mathbf{T}_{\mathbb{D}}(\text{stmt}, \mathcal{E}, D)$ is a sound over-approximation of the non-erroneous concrete states in $\llbracket \text{stmt} \rrbracket(\gamma_{\mathbb{D}}(D))$. Moreover, an alarm is emitted if $\Omega \in \llbracket \text{stmt} \rrbracket(\gamma_{\mathbb{D}}(D))$. \square

Theorem 12. Let stmt be a copy $*_{\tau}a_1 \leftarrow *_{\tau}a_2$, and let D be an abstract state of the domain \mathbb{D} . If $\llbracket e \rrbracket_D^\# : \mathbb{E} \rightarrow \mathbb{E} \times \mathbb{A}$ is an evaluation function satisfying the properties of proposition 1, then the function $\llbracket \text{stmt} \rrbracket^\#$ defined as follows is a sound transfer function on non erroneous states, according to definition 8.

$$\begin{aligned} \llbracket \text{stmt} \rrbracket^\#(D) &= \mathbf{T}_{\mathbb{D}}(\text{stmt}, \mathcal{E}, D) && \text{where } \mathcal{E}, \mathbf{A} = \llbracket a_1 \rrbracket_D^\#(\mathcal{E}') \\ &&& \text{and } \mathcal{E}', \mathbf{A}' = \llbracket a_2 \rrbracket_D^\#(\top_{\mathbb{E}}) \end{aligned}$$

Moreover, if it exists a concrete state S in $\gamma_{\mathbb{D}}(D)$ for which the execution of stmt fails (i.e. such that $\llbracket \text{stmt} \rrbracket(S) = \Omega$), then \mathbf{A} or \mathbf{A}' is a non-bottom alarm map, and an alarm is thus emitted by the analyzer (EVA emits all alarms of $\mathbf{A} \sqcup_{\mathbb{A}} \mathbf{A}'$ that have a non-true status).

Proof. The same reasoning as for theorem 11 applies. \square

It is worth noting that the dereference of the right address is not evaluated; indeed, in the concrete semantics of a copy, the bytes located at the right address are only copied at the location denoted by the left address, without further computation.

Theorem 13. Let stmt be an assumption $e == 0?$, and let D be an abstract state of the domain \mathbb{D} . If $\llbracket e \rrbracket_D^\# : \mathbb{E} \rightarrow \mathbb{E} \times \mathbb{A}$ and $\llbracket \leftarrow e \rrbracket_D^\# : \mathbb{E} \rightarrow \mathbb{V}^\# \rightarrow \mathbb{E}$ are a forward and a backward evaluation functions satisfying the properties of propositions 1 and 2, then the function $\llbracket \text{stmt} \rrbracket^\#$ defined as follows is a sound transfer function on non erroneous states, according to definition 8.

$$\begin{aligned} \llbracket \text{stmt} \rrbracket^\#(D) &= \mathbf{T}_{\mathbb{D}}(\text{stmt}, \mathcal{E}, D) && \text{where } \mathcal{E} = \llbracket \leftarrow e \rrbracket_D^\#(\mathcal{E}', 0^\#) \\ &&& \text{and } \mathcal{E}', \mathbf{A} = \llbracket e \rrbracket_D^\#(\top_{\mathbb{E}}) \end{aligned}$$

Moreover, if it exists a concrete state S in $\gamma_{\mathbb{D}}(D)$ for which the execution of stmt fails (i.e. such that $\llbracket \text{stmt} \rrbracket(S) = \Omega$), then \mathbf{A} is a non-bottom alarm map, and an alarm is thus emitted by the analyzer (EVA emits all alarms of \mathbf{A} that have a non-true status).

Proof. Let S be a concrete state of $\gamma_{\mathbb{D}}(D)$. The semantics of assignment, given in Figure 4.9, defines $\llbracket \text{stmt} \rrbracket(S) = \Omega$ only if $\llbracket e \rrbracket^\Theta(S) = \Omega$. Otherwise, the execution of stmt in S succeeds or blocks.

- Case 1. $\llbracket e \rrbracket^\Theta(S) = \Omega$. By lemma 5, $\exists A \in \text{alarms}(e)$, $\neg A(S)$.
 As $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \top_{\mathbb{E}})$, the soundness of the evaluation (proposition 1) ensures that $A' \models_{\mathbb{A}} \llbracket e \rrbracket^\Theta(S)$. By definition 38, $A(A) \neq \text{true}$. Therefore, A is not the bottom alarm map, and the alarm A is emitted by the analyzer.
- Case 2. $\llbracket e \rrbracket^\Theta(S) \neq \Omega$ but $\llbracket e \rrbracket^\Theta(S) \neq 0$.
 Then $\{\text{stmt}\}(S) = \emptyset \subseteq \gamma_{\mathbb{D}}(\mathbf{T}_{\mathbb{D}}(\text{stmt}, \mathcal{E}, D))$.
- Case 3. $\llbracket e \rrbracket^\Theta(S) = 0$
 By proposition 1, we have $S \in \gamma_{\mathbb{E}}(\mathcal{E}')$.
 By proposition 2, we also have $S \in \gamma_{\mathbb{E}}(\mathcal{E})$.
 Finally, $S \in \gamma_{\mathbb{D}\mathbb{E}}(D, \mathcal{E})$ and definition 59 ensures that:

$$\{\text{stmt}\}(S) \in \gamma_{\mathbb{D}}(\mathbf{T}_{\mathbb{D}}(\text{stmt}, \mathcal{E}, D))$$

Finally, $\mathbf{T}_{\mathbb{D}}(\text{stmt}, \mathcal{E}, D)$ is an sound over-approximation of the non-erroneous concrete states in $\{\text{stmt}\}(\gamma_{\mathbb{D}}(D))$. Moreover, an alarm is emitted if $\Omega \in \{\text{stmt}\}(\gamma_{\mathbb{D}}(D))$. \square

7.3.2 Domain Product

The product of several abstract domains \mathbb{D}_1 to \mathbb{D}_n is straightforward. An abstract state is a tuple $(D_1, \dots, D_n) \in \mathbb{D}_1 \times \dots \times \mathbb{D}_n$. The lattice structure is that of the direct product (see Section 2.3.2). In particular, the product maintains the properties of the underlying lattices. If the domains \mathbb{D}_1 to \mathbb{D}_n are complete lattices, then their product is a complete lattice as well. The forward and backward queries of the product perform the meet of the abstractions produced by each domain. They have been formally defined and proved correct all through the definition of queries, in section 7.1 and 7.2. Finally, the abstract transformers of the product are the component-wise applications of the abstract transformers of each underlying domain.

$$\begin{aligned} \mathbf{T}_{\mathbb{D}_1 \times \dots \times \mathbb{D}_n}(\text{stmt}, \mathcal{E}, (D_1, \dots, D_n)) &\triangleq (\mathbf{T}_{\mathbb{D}_1}(\text{stmt}, \mathcal{E}, D_1), \\ &\dots, \\ &\mathbf{T}_{\mathbb{D}_n}(\text{stmt}, \mathcal{E}, D_n)) \end{aligned}$$

The monotonicity of the transfer functions requires the monotonicity of the computation of the valuation. This requires the monotonicity of the abstract semantics of value and location abstractions, as well as the monotonicity of all domain queries. However, as most analyzers (see Section 2.2.6), EVA relies mainly on widening operators to ensure a fast convergence of its analysis. The monotonicity of transfer functions, as a complete lattice structure of domains, are ancillary.

The interactions between domains are entirely embedded in the collaborative computation of the valuation. The same valuation is

1	<code>int main (int i) {</code>
	<code> int x = 17;</code>
	<code> int t[] = {1, 2, 3, 4, 5};</code>
	<code> •</code>
5	<code> int r = t[i] + x;</code>
	<code> if (r + i >= 20)</code>
	<code> [...]</code>

	Environment	Array domain
States D	$i \mapsto \top$	$t : [1; 2; 3; 4; 5]$
	$x \mapsto [17]$	
	$r \mapsto \perp$	

Figure 7.8: Interpreting a statement

then used separately by each domain for interpreting the statement as precisely as possible. The examples of this chapter are intended to illustrate how domains contribute cooperatively to the computation of a precise valuation, but also how this valuation supports their interpretation of statements. Chapter 8 presents the domains currently available in [EVA](#), and their different uses of the valuation. Finally, the next section identifies the relevance of some elements of a valuation owing to the reductions that have been performed during the evaluation.

7.3.3 Tracking Reductions

Some abstractions included in the valuation given to a domain transformer are especially relevant to assist the interpretation of a statement. This is for instance the case for the value abstraction of the right expression of an assignment, and for the location abstraction of the address being assigned. However, the valuation also contains sound approximations of the values or locations of all subterms of the evaluated expressions and addresses, and these abstractions may also help a domain to be more accurate. To process a conditional test, the abstractions computed for the subterms of the condition can be especially important, as the backward propagation of the truth value of the condition could have reduced them.

Example 30. Let us consider the analysis of the code of Figure 7.8, with the previously introduced environment and array domain. The table represents their abstract states at line 4. The interpretation of the statement at line 5 starts by evaluating the expression $t[i] + x$, that leads to the interval $[18..22]$. This interval can be directly used by the environment domain as an abstraction of the possible values of r after the assignment. Furthermore, the evaluation of $t[i]$ raises an alarm asserting that the success of the dereference requires i to be

between 0 and 4. And the backward propagation on $t[i]$ reduces the abstraction of i to the interval $[0..4]$.

This new interval is a sound abstraction of the possible values of i after the assignment, as any other concrete value leads to the error value for the dereference $t[i]$ at line 5 —and thus aborts the program execution in our semantics, or exhibits an undefined behavior according to the C standard. The environment domain can use the abstractions of $t[i]+x$ and i to compute the abstract state:

$$\begin{aligned} i &\mapsto [0..4] \\ x &\mapsto [17] \\ r &\mapsto [18..22] \end{aligned}$$

which is a sound and precise abstraction of the concrete states after the assignment of line 5.

Afterwards, the interpretation of the conditional statement at line 6 propagates the value $1^\#$ to the condition $r+i \geq 20$. The backward propagators of the interval semantics reduce the abstraction of r to $[20..22]$, and cannot reduce the abstraction $[0..4]$ of i . Without interpreting the comparison, the environment domain can use again the resulting valuation to compute the abstract state:

$$\begin{aligned} i &\mapsto [0..4] \\ x &\mapsto [17] \\ r &\mapsto [20..22] \end{aligned}$$

which is a sound and precise abstraction of the concrete states at the beginning of the *if* branch at line 5.

Although some abstractions of the valuation may assist a domain transformer, some others are already known by the abstract state. For instance, at line 5 of the previous example, the value abstraction of x in the valuation is the interval $[17]$ provided by the environment domain. Thus, the domain transformer cannot learn anything new from this abstraction. Similarly, the value abstraction of i is not reduced by the backward propagation of line 6, and the abstract state already contains it. Generally, browsing the full valuation seeking new or more precise abstractions to refine an abstract state can be expensive, especially when complex expressions are involved in a statement. Updating the state for each abstraction of the valuation is obviously too costly, and determining which abstractions can refine the state requires a read of the state and a comparison for each stored abstraction.

To address this issue, the generic evaluator of [EVA](#) tracks the reduction of the value abstractions supplied by the abstract domains during the backward propagation, and highlights the abstractions that have actually been reduced. In practice, the valuation stores abstractions

along with a boolean flag indicating whether the abstraction is more precise than the one supplied by the domain for the given expression. This boolean is false by default, and is updated only when a reduction is achieved. A current limitation is that [EVA](#) tracks reductions for the whole value abstraction, and not for each component. Thus, as soon as a component is reduced, all components are marked as reduced.

A value abstraction provided by a domain product $\mathbb{D}_1 \times \mathbb{D}_2$ can be reduced by a backward propagation (which is tracked by the evaluator), but the value v_1 provided by a specific domain \mathbb{D}_1 can also be reduced when performing the meet with the value v_2 provided by the other domain \mathbb{D}_2 . This kind of reduction is tracked internally by the domain combiner of [EVA](#), which updates accordingly the boolean flags for each domain.

7.3.4 *Related Works and Limitations*

The communication through value abstractions shares some objectives and aspects of an open product (see Section 2.3.4). The queries of both designs fulfill the same role: they express the properties inferred by a domain in a form that another domain can use. Moreover, the mutual refinement of abstract domains is not postponed to the end of transfer functions: the information stemming from queries is available to the domains during their interpretation of a statement. However, the open product limits queries to boolean functions, without further restrictions. Our design both extends the type and defines more clearly the scope of the queries: they abstract the possible values of addresses and expressions through specific abstractions (values, locations and alarms). Furthermore, the transfer functions do not interact directly with the queries, but with the result of complete evaluations of addresses and expressions. The evaluation engine, described in Chapter 6, exploits all queries and includes various mechanisms to achieve a better precision. This allows for further and easier interactions between domains. In particular, a forward propagation allows a query about a subterm to refine the abstraction of an expression; a backward propagation allows a query about an expression to refine the abstractions of the subterms. In this way, a domain may take advantage of a property about an expression that has been computed from queries on other expressions. Finally, the oracle (Section 7.1.3) and the reducer (Section 7.2.2) provides access to the complete forward and backward evaluations, that cannot be implemented as a query by a single domain.

We now focus more specifically on the comparison between our model of domains interactions and the communication by messages introduced by the Astrée analyzer (see Section 2.3.6).

7.3.4.1 *Architecture*

The Astrée design requires the communication system to be maintained in parallel of an abstract semantics implementation. The multiple channels conveying messages can be invasive in the analyzer architecture. In [EVA](#), the interface between abstract domains consists of alarm, value and location abstractions, that are part of the abstract semantics implemented by the analyzer. We believe that separating value and state abstractions is a very convenient way to structure an abstract interpreter. It naturally allows for the implementation of smaller modules connected by clear interfaces. Moreover, while some of the communication channels of Astrée are oriented, our product of domains is unordered: the combination order does not impact the precision of the analysis.

7.3.4.2 *Expressiveness*

The communication by messages is unquestionably more expressive than the interactions through values and locations. The set of messages is extensible, and any property can be as expressed as a new kind of message. On the other hand, the abstract domains of [EVA](#) only interact through abstractions of address locations and expression values. Nevertheless, we claim that our design enables the information exchanges that have been actually implemented via messages in Astrée and Verasco. Since the messages of Astrée [\[Cou+06\]](#) are not as thoroughly detailed as those of Verasco [\[Jou+15; Jou16\]](#), we focus on the latter analyzer. For each kind of message, we describe how to achieve the same interactions between the domains of [EVA](#):

- the messages expressing an interval, a congruence relation or the exact value of a variable are naturally supported by our value abstractions, which include intervals, arithmetical congruence and small sets of exact values.
- the messages expressing the equality between two expressions can be replaced by the use of the oracle or the reducer, to state that two expressions evaluates to the same value. This has been illustrated by examples [26](#) and [29](#).

Finally, the emission of the alarms in [EVA](#) involves directly all the abstractions of our analyzer. Some abstractions can thus be dedicated to removing specific alarms without having to propagate information towards existing domains. This cannot be achieved by a standard communication by messages.

7.3.4.3 *Ease of Use*

We believe that the design we propose facilitates the introduction of new domains —and new interactions between them— in the analyzer.

First, it seems natural to build an abstract domain upon value and location abstractions. In Astrée, the intervals are the main actor of the communication between abstract domains. In EVA, numerical domains use the location abstractions to interpret precisely dereferences and assignments through pointers, without embedding an alias analysis. In general, we have shown how our design allows the precise combination of domains that handle different subsets of the language semantics and of the undesirable behaviors tracked by the analyzer. Provided that the union of domains covers the whole semantics, the evaluation engine makes the connection between them and even enables relevant mutual refinements between them (see examples 24 and 27).

Second, the value and location abstractions are natural candidates for a reduced product. This enables interactions even between domains understanding different components of the product, as demonstrated by example 6. By mutually refining the components of the value or location abstractions, new communications can be established between existing domains without having to modify them. Moreover, domains need not be adapted when a new kind of value is added. The idea is already exploited in EVA, where the values are a reduced product between interval and arithmetical congruence.

Third, the oracle and the reducer allow a relational domain to fully exploit its inferred relations with the seamless support of all others domains, without resorting to an explicit communication channel. A relation between expressions is expressed as a comparison between the result of their evaluations, and the evaluation engine harnesses automatically the information known by all domains. This spares non-relational domains to process relational properties, while taking advantage from them.

7.3.4.4 *Limitations*

The interactions between EVA's domains are currently limited to the transfer functions. The inclusion, join and widening operations do not embed any communication mechanism. A simple and suitable solution in our architecture would be to provide the domains with the oracle and the reducer when performing these operations. A domain could then request the value of an expression to be more accurate, or state the value of an expression to refine the other domains. However, this would make the signature of these functions much more complex: as binary operations, they would require two oracles (one for each incoming state) and one reducer (for the resulting state). As we seek a clear interface to ease the implementation of new abstract domains, we chose to limit the interactions to the transfer functions, where the relevant evaluations to perform are more obvious.

The use of a single valuation as means of communication can also be seen as a limitation. Astrée and Verasco embed in parallel channels

for preconditions (whose messages are emitted by the states before the transfer function) channels for postconditions (whose messages are incrementally emitted by the states resulting from the transfer function). Although there is no such distinction in our design, the meaning of the valuation received by a transfer function stands in between preconditions and postconditions. The valuation results from evaluations. While these evaluations are performed in the abstract states before the considered statement, the resulting valuation is computed assuming that the concrete evaluations succeed (thanks to the alarm maps reporting the error cases). For instance, the location abstraction of an assignment address has been reduced to its valid parts for a write operation. The value abstractions of variables can have been reduced according to emitted alarms (see examples 11 and 12). Furthermore, the valuation for a test statement has been reduced by assuming the condition of the test. Therefore, the valuation also contains postconditions of the statement.

Finally, value and location abstractions remain less expressive and less extensible than arbitrary messages. The structuring of datatype combinations, described in Section 3.3, could help to overcome the limitations of our communication system. It allows interacting directly with the components of a generic product. Two specific domains can thus be interconnected, should they require a novel form of communication that cannot be encoded through abstractions of expressions. While the evaluation to values and locations organizes seamless interactions between domains, the structure of the domains product enables explicit communications between them. However, this comes down to an approximate reduced product, specific to the domains it involves. Until now, we had not experiencing any problem requiring such a radical workaround to our analyzer design.

This chapter presents the different abstract domains that have been successfully implemented within [EVA](#). None of them is a novelty *per se*. The main domain is an adaptation of the abstract semantics implemented by the former abstract interpreter of FRAMA-C, the [VALUE](#) plugin. The others are well-known abstractions of the existing literature. Nevertheless, their diversity is worth noticing: some are numerical, some are symbolic, some are relational; some abstract the memory and others do not. They also includes a binding to the existing domains provided by the [APRON](#) library. Being able to embed such different abstractions easily into [EVA](#)—five new domains have been introduced in less than a year—seems a good validation of our design choices.

This chapter does not provide a detailed, formal or exhaustive description of the implemented domains. It mainly underlines how these domains fit into the [EVA](#) architecture. It also compares the performance and the precision between [VALUE](#) and [EVA](#), when the latter is instantiated with abstractions that are equivalent to the semantics implemented by the former. It finally presents some experimental results obtained with the newer abstractions.

8.1 THE CVALUE DOMAIN

The most important abstract domain of [EVA](#) is the `CVALUE` domain—the name is historical. It is inherited from [VALUE](#), and was retrofitted for [EVA](#). It is a non-relational domain describing whole memories as maps from variables to sequences of sliced abstract values.

8.1.1 Description

`CVALUE` embeds directly the standard value abstractions of [EVA](#), described in Section 5.3. Its state domain is quite involved, and we refer the reader to [Kir+15, §4] for a more complete explanation. The memory is represented as a map from the variables of the program to *offsetmaps* [BC11]. Offsetmaps are basically maps from bits-expressed intervals of offsets to abstract values. A struct with two int fields containing the abstract values v_1 and v_2 respectively is represented, on a 32-bit architecture, by

$$\langle 0 - 31 \rangle \rightarrow v_1 \quad \langle 32 - 63 \rangle \rightarrow v_2 \quad (8.1)$$

The abstract values are the cvalue abstractions of Section 5.3, plus two additional booleans that abstract the possibility that the value may be uninitialized, or a dangling pointer. Most ocaml datastructures are hash-consed for efficiency [CDo8].

The memory is untyped, and it is possible to write an abstract value of any type anywhere in the memory. This makes the domain able to represent efficiently and precisely both low-level concepts such as unions and bitfields, or high-level ones, such as arrays. Thus, the offsetmap 8.1 can also be the representation of an array of two cells containing respectively the abstract values v_1 and v_2 ; or the representation of a 64-bits variable accessed as an array of two 32-bits cells. Assignments overlapping existing bindings are automatically handled, and remain precise. Assignments to a very large number of non-contiguous locations are automatically approximated. This domain shares some similarities with Miné’s [Mino6a]. However, assignments with partial overlap to existing bindings are handled quite differently, as Miné’s methodology may store multiple bindings at the same offset.

As the cvalue abstractions embed some precise representations of pointer values, the CVALUE domain is able to infer accurate alias information. This drastically simplifies the writing of other domains, that do not need to track information about pointers. Currently, CVALUE is the only domain featuring an alias analysis, and the other domains rely on it to interpret the assignments involving pointers.

Let us add that the abstractions of CVALUE (the offsetmaps, as well as the maps from variables) are written using generic ocaml functors, which are also functors in the abstract interpretation sense. We have reused these abstractions when writing some of the new EVA domains.

8.1.2 Integration in the EVA Framework

8.1.2.1 Participation to the Cooperative Evaluation

A CVALUE state is able to represent all objects in memory, from scalar variables to arrays, unions and structures of any types. During the cooperative evaluation of an expression, the CVALUE domain provides a forward and backward abstract semantics of dereference for the cvalue abstractions. To the other queries $F_{\mathbb{D}}^{\#}$ concerning arbitrary expressions, the domain always returns the top value and alarm abstractions. It does not use the oracle nor trigger new reductions.

8.1.2.2 Interpretation of Statements

The CVALUE domain uses the valuation produced on any statement to refine its internal state. It browses the valuation to collect the dereferences: as the domain gathers information on variables only, the

abstractions computed for other expressions cannot directly help to reduce its state. More specifically, the domain searches the dereferences such that:

- the value abstraction is flagged as reduced compared to the value abstraction provided by the dereference semantics of the current CVALUE state. Otherwise, the cvalue abstraction stored in the state is already as precise as the one in the valuation.
- the location abstraction of the address represent only one concrete location (this location abstraction is also stored in the valuation). Otherwise, the cvalue abstraction cannot be soundly written in the state, since its exact location remains unknown.

For such dereferences $*_{\tau}a$, the value abstraction stored in the valuation is written at the single possible location of the address a .

On a conditional statement `if(c)`, the produced valuation is an abstraction of the set of concrete states before the statement in which the evaluation of the condition c succeeds *and* returns a non-zero value. Thanks to the backward propagation achieved by the cooperative evaluation, updating the state from the valuation allows the domain to take into account that the condition holds without having to understand it.

On an assignment, the produced valuation is a sound abstraction of the concrete states before the statement in which the evaluations of the left address and of the right expression succeed. As the abstract transformer has to interpret the statement only for those states, it starts by updating the state from the valuation. This allows the domain to gain some precision, by rejecting some concrete states leading to an error at this program point. Then, the transformer still has to interpret the assignment. It uses the location abstractions cooperatively computed for the addresses. This allows it to interpret in the same precise way the assignments from both low-level and high-level view of the memory. The interpretation of an assignment $*_{\tau}a := e$ amounts to write the cvalue abstraction computed for e at the concrete locations represented by the location abstraction computed for a . If the location abstraction represents a single concrete location, this operation overwrites the previous value stored by the state at this location. Otherwise, the assignment can write at any of the concrete locations described by the address abstraction, and let the others unmodified. For each of these possible locations, the CVALUE domain stores the join between the cvalue abstraction of e and the previous cvalue abstraction stored at this location. The offsetmap data structure handles automatically the write operation in both cases.

On a copy, the CVALUE domain uses the location abstractions computed for both left and right addresses. It copies and pastes the offsetmap slice from the right locations to the left locations, performing

joins if necessary (i.e. if the location abstractions have non-singleton concretizations).

8.1.3 *Performance Compared to VALUE*

As the CVALUE domain has the same abstract semantics as the one which was implemented by the former abstract interpreter [VALUE](#), it is especially worthwhile to compare the performance between [VALUE](#) and [EVA](#), when the latter is instantiated with this domain alone. Structuring the abstract interpreter into multiple separate layers interacting through minimal interfaces was mandatory to achieve a generic analyzer, but also entails some slowdown of the analysis. Many optimizations of the [VALUE](#) analyzer relied on the tight connection between the analyzer and the implemented semantics. Some had to be abandoned within [EVA](#) for the sake of genericity and extensibility; others have been kept thanks to the [GADT](#) structure of the domain combination: this mechanism, detailed in Section 3.3, allows the generic abstract interpreter to extract the cvalue component from an arbitrary product of abstraction and to perform the optimizations accordingly. If the product of combination does not include a cvalue component, then these optimizations are disabled.

Furthermore, the evaluation strategy has been entirely revised in [EVA](#). In particular, the complete and systematic backward propagation made by [EVA](#) on every evaluated expression is new; formerly, [VALUE](#) relied mainly on syntactic patterns to perform some backward propagations. Even if all backward propagators have not been implemented yet in the cvalue abstractions —backward propagators for multiplication and division are especially lacking—, the new propagators are strictly stronger than before, and the generic evaluator resorts to them more aggressively. This has improved the precision of [EVA](#) with respect to [VALUE](#).

Table 8.1 presents the results of the experiments we conducted to compare the performance between [VALUE](#) and [EVA](#). During the development of the [EVA](#) interpreter, we continuously validated it on the pre-existing test suite of [VALUE](#). These tests can be found in the tests/value directory of FRAMA-C. They are C code snippets intended for verifying the soundness or the accuracy of the analyzer on particular constructs. We used them to ensure that the new analyzer had no regression, whether in terms of soundness or precision. Table 8.1a shows the number of files that bring out differences between the two interpreters. We have omitted two tests that focus on recursion and downcasts. On the one hand, [VALUE](#) features a preliminary support for recursive functions, which has not been ported on [EVA](#) yet. On the other hand, [EVA](#) emits alarms about unsafe downcasts, and [VALUE](#) does not. On all other tests, not only [EVA](#) exhibits no regression with respect to the former analyzer, but it also obtains a

tests/value files	389
Precision loss in EVA	0
Less emitted alarms in EVA	7 (1.8%)
Reduced final state in EVA	22 (5.6%)
Precision gain with EVA	27 (6.9%)

(a) Improved precision on the test suite of FRAMA-C

	loc	statements	VALUE	EVA	
idct	342	586 (94%)	1.5s	1.4s	time
			56	56	alarms
tweetnacl	799	658 (99%)	9.8s	11.9s	time
			2	2	alarms
hiredis	3935	982 (83%)	10.3s	13.2s	time
			212	208	alarms
papabench	3199	1845 (55%)	1.1s	1.3s	time
			31	31	alarms
debic	5326	2872 (98%)	9.3s	10.9s	time
			62	55	alarms
gzip	5196	4170 (94%)	81s	94s	time
			1928	1865	alarms
polarssl	24k	2758 (49%)	227s	236s	time
			159	158	alarms
indus1	99k	30955 (81%)	408s	430s	time
			228	228	alarms
indus2	125k	36782 (80%)	758s	715s	time
			244	244	alarms

(b) Performance comparison on case studies

Table 8.1: Comparisons between [VALUE](#) and [EVA](#)

better result on 27 of the 389 test files of FRAMA-C. A better result means either less alarms (for 7 files) or a more precise abstract state at the end of the main function (for 22 files). Note that these tests have been designed for the legacy interpreter, and not for the new features of EVA. It is thus quite positive to register improvements within this test suite, without even introducing new abstractions.

Table 8.1b compares the performance of both analyzers on case studies. For each case study, the table indicates the number of lines of code (without blanks, comments and headers), the number of statements visited by the analysis (with the coverage percentage for the whole program), and the analysis time and number of alarms emitted for each analyzer. The first line concerns the biggest file of the test suite of FRAMA-C, which is not in the previously tested tests/value directory, but in tests/idct. The six following lines present various case studies on open-source codes, mainly used to benchmark the analyzer. The analysis is more or less well configured between those codes, which leads to a large disparity in the number of emitted alarms (but both analyzers always run with the same parameters for a given case study). The last two lines are about proprietary softwares. They are safety-critical embedded programs on which EVA is currently deployed in industry.

EVA shows improvements in 4 of these 9 case studies, emitting less alarms than VALUE did. However, these improvements remain small, except for the *debie* program, where the number of emitted alarms falls from 62 to 55 (a decrease of 11%). At the same time, EVA is generally slower than VALUE by 5 to 20%, which remains satisfactory. The biggest time increase happens on *tweetnacl* and *hiredis*, where the analyzers are respectively 21% and 28% slower. However, these analyzers take about 10 seconds only. On *polarssl* and *indus1* where the analyzers take several minutes, EVA is only 5% slower. On our biggest benchmark, EVA is even faster than VALUE from 6%. This may be due to more precise abstract states, leading to a faster convergence.

8.1.3.1 Efficiency of Possible Evaluation Strategies

Chapter 6 was dedicated to the evaluation of expression in EVA. In particular, Section 6.2.3 presented the strategy of intertwining propagations used in EVA, and discussed some alternatives. Table 8.2 presents the experimental results obtained for different strategies:

- EVA: the current strategy used in EVA, described in Section 6.2.3, consists in a single complete backward propagation after any forward evaluation.
- (-): expressions and addresses of assignments are only forward evaluated, without any backward propagation. It is worth noting that the reduction of a dereferenced location to its valid part is done during the forward evaluation, and thus still achieved

	(-)	EVA	(+)	(++)
Differences on tests/value files	40	-	0	1

(-) : no backward propagation

EVA : one forward and one backward propagations

(+) : second forward propagation

(++) : second forward and second backward propagations

	stmts	(-)	EVA	(+)	(++)	
tweetnacl	658	11.9s	11.9s	12.5s	13.1	<i>time</i>
		2	2	2	2	<i>alarms</i>
hiredis	982	12.9s	13.2s	13.4s	14.0	<i>time</i>
		211	208	208	208	<i>alarms</i>
papabench	1845	1.26s	1.27s	1.31s	1.33s	<i>time</i>
		31	31	31	31	<i>alarms</i>
debic	2872	10.9s	10.9s	11.3s	11.7	<i>time</i>
		57	55	55	55	<i>alarms</i>
gzip	4183	83s	94s	96s	103s	<i>time</i>
		1923	1865	1859	1859	<i>alarms</i>
polarssl	2758	228s	236s	239s	?	<i>time</i>
		158	158	158	158	<i>alarms</i>

Table 8.2: Comparisons between different evaluation strategies

in this configuration. The backward propagation of the truth value of a condition is still performed.

- (+): a second forward propagation is systematically performed after any backward propagation.
- (++): a second backward propagation is systematically done after the second forward propagation.

For each strategy, the first table shows the number of differences on the test suite of [VALUE](#), and the second table presents the analysis time and the number of emitted alarms on the open-source case studies previously used. These experiments have been conducted with the CVALUE domain only.

Omitting the backward propagation entails a precision loss in 40 of the 391 test files of tests/value —characterized by either more emitted alarms or a less precise final abstract state. This also results in a few more alarms in 3 of the 6 open-source case studies for a small reduction of the analysis time. Interestingly, in this configura-

tion, [EVA](#) obtains approximately the same results and analysis times than [VALUE](#) on the biggest open-source case studies, *gzip* and *polarssl*. It is still slightly more precise thanks to the stronger backward propagation on conditionals.

On the other hand, the second forward propagation and the second backward propagation show nearly no difference in the test suite or in the open-source case studies. The only difference in the test suite comes from a non-idempotent backward propagator, which gain precision by being applied twice. The only difference in the case studies occurs on *gzip*, where the second forward propagation avoids the emission of 6 alarms out of 1865. Even if the cost of these additional propagations is insignificant, their low impact on the analysis precision seems to make them unnecessary.

8.2 THE EQUALITY DOMAIN

The `CVALUE` domain provides a precise representation of memories, but is unable to infer relations between variables. The first relational abstraction introduced in [EVA](#) has been a symbolic domain tracking Herbrand equalities between C expressions. Our intentions was somewhat similar to those of Miné [[Mino6b](#)], in particular abstracting over temporary variables resulting from code normalization. Our equality domain is especially interesting because of its use of the oracle and the reducer during the cooperative evaluation. The equalities are deduced from equality conditions and from assignments that do not read the memory locations being modified. They are then used to trigger new forward evaluations or backward propagations. A sound interpretation of a statement also requires to remove from the abstract state the equalities which hold no longer afterwards. This is only needed on assignments, for the equalities between expressions that depend on the written location. To do so, the equality domain needs an abstraction of the byte locations on which an expression depends.

This section first defines the dependences of expressions and addresses, as well as abstractions of these dependences. It then formalizes the abstract states of the domain, its queries and its abstract transformers. Finally, it presents some details about the implementation and the experimental results obtained with this new domain.

8.2.1 Dependences of an Expression

We first define the dependence of an expression as an operator `deps` computing sets of bytes in $\mathcal{P}(\mathcal{L}_{\text{bytes}})$: `deps(e , S)` is the set of the memory bytes on which the evaluation of e depends in the concrete state S . We remind the reader of the definition [25](#) of memory locations: for

a pointer value l , $\text{loc}_\tau(l)$ is the set of $\text{sizeof}(\tau)$ bytes in memory starting at l .

$$\text{loc}_\tau(\&x, i) \triangleq \{(\&x, i + n) \mid 0 \leq n < \text{sizeof}(\tau)\}$$

Definition 60 (Dependences). If the evaluation of an expression e succeeds in a concrete state S , the dependence of e in S is inductively defined as:

$$\begin{aligned} \text{deps}(*_\tau a, S) &\triangleq \text{loc}_\tau(l) \cup \text{deps}(a, S) \\ &\quad \text{where } \forall \theta \in \Theta_P, \llbracket a \rrbracket_{S(\theta)} = l \in \mathcal{L}_\tau \\ \text{deps}(\diamond(e_1, \dots, e_n), S) &\triangleq \bigcup_{i \in \{1, \dots, n\}} \text{deps}(e_i, S) \\ \text{deps}(cst, S) &\triangleq \emptyset \\ \text{deps}(t[i], S) &\triangleq \text{deps}(t, S) \cup \text{deps}(i, S) \\ \text{deps}(e.\text{field}, S) &\triangleq \text{deps}(e, S) \end{aligned}$$

Let us recall that if the evaluation of a dereference $*_\tau a$, then the concrete semantics guarantees the existence of a valid location l such that in all layouts θ , $\llbracket a \rrbracket_{S(\theta)} = l$.

Lemma 23. Let e be an expression and S a concrete state such that the evaluation of e in S succeeds: $\llbracket e \rrbracket^\Theta(S) = V$. The resulting concrete value V only depends on the bytes of $\text{deps}(e, S)$. If S' is another concrete state that coincide with S on these bytes, then $\llbracket e \rrbracket^\Theta(S') = V$.

$$\begin{aligned} \forall S, S' \in \mathcal{S}, \forall e \in \text{expr}, \llbracket e \rrbracket^\Theta(S) \neq \Omega, \\ (\forall \theta \in \Theta_P, \forall l \in \text{deps}(e, S), S(\theta)(l) = S'(\theta)(l)) \\ \Rightarrow \llbracket e \rrbracket^\Theta(S') = \llbracket e \rrbracket^\Theta(S) \wedge \text{deps}(e, S) = \text{deps}(e, S') \end{aligned}$$

Proof. The concrete semantics of expressions involves the concrete states only on dereferences, where the bytes of the address are read. On two states coinciding on these bytes, the concrete semantics leads to the same concrete values as result of the evaluation.

The dependence of an expression e in a concrete state depends only on the evaluation of the addresses a dereferenced in the expression. The evaluation of such an address a depends on the bytes of $\text{deps}(a, S)$, and $\text{deps}(a, S) \subseteq \text{deps}(e, S)$. Thus, the evaluation of a is the same on two states coinciding on these bytes, and thus the dependences of e is the same on such states. \square

Definition 61 (Zone abstraction). An abstraction of arbitrary sets of bytes (i.e. dependences), called zone, is a collection of abstractions $\mathbb{Z}^\#$ equipped with:

- a concretization $\gamma_{\mathbb{Z}} : \mathbb{Z}^\# \rightarrow \mathcal{P}(\mathcal{L}_{\text{bytes}})$. A zone abstraction z is a sound approximation of the dependence of an expression e in a concrete state S if and only if $\text{deps}(e, S) \subseteq \gamma_{\mathbb{Z}}(z)$.

- a lattice structure $(\mathbb{Z}^\#, \sqsubseteq_{\mathbb{Z}}, \sqcup_{\mathbb{Z}}, \sqcap_{\mathbb{Z}}, \top_{\mathbb{Z}})$ fulfilling the properties specified in Figure 2.4 according to the concretization.
- an injection $\text{zone} : \mathbb{L}^\# \rightarrow \text{type} \rightarrow \mathbb{Z}^\#$ from location abstractions, such that for a location abstraction l and a type τ , all bytes of $\text{loc}_\tau(\gamma_{\mathbb{L}}(l))$ are in the concretization of $\text{zone}(l, \tau)$.

$$\forall l \in \mathbb{L}^\#, \forall \tau, (\lambda_.L) \in \gamma_{\mathbb{L}}(l) \Rightarrow \text{loc}_\tau(L) \subseteq \gamma_{\mathbb{Z}}(\text{zone}(l, \tau))$$

We assume given a zone abstraction $\mathbb{Z}^\#$. In the EVA equality domain, the zone abstractions are the maps from variables to integer offset abstractions presented in Section 5.3.1.3. However, the formalization below does not depend on the chosen zone abstraction.

8.2.2 The Equality Abstract States and Queries

We can now define the states and the semantics of the equality domain. An abstract state of this domain is a set of equalities between syntactic expressions, and represents all the concrete states in which the equalities hold. An abstraction of the dependence of each expression involved in an equality is also computed by the domain.

Definition 62 (Equality domain). The equality domain stores expressions along with an abstraction of their dependences. An equality in \mathbb{E} is denoted as a set of expressions. An abstract state in \mathbb{D}_{eq} is a set of equalities. The concretization of such an abstract state is the set of concrete states in which, for each equality:

- all the expressions evaluate to the same concrete value;
- the zone stored for each expression is a sound approximation of its dependences.

$$\begin{aligned} \mathbb{E} &\triangleq \mathcal{P}(\text{expr} \times \mathbb{Z}^\#) \\ \mathbb{D}_{\text{eq}} &\triangleq \mathcal{P}(\mathbb{E}) \\ \gamma_{\text{eq}} : \mathbb{D}_{\text{eq}} &\rightarrow \mathcal{S} \end{aligned}$$

$$S \in \gamma_{\text{eq}}(D) \Leftrightarrow \forall E \in D, \exists V \in \mathcal{V}, \forall (e, z) \in E, \begin{cases} \llbracket e \rrbracket^\Theta(S) = V \\ \text{deps}(e, S) \subseteq \gamma_{\mathbb{Z}}(z) \end{cases}$$

The equality domain should also maintain some invariants to make the handling of abstract states easier. The equalities should be disjoint sets of expressions, and no equality should be a singleton, as a singleton equality brings no information whatsoever. These conditions do not impact the concretization of an abstract state, and thus do not impact the soundness of the domain and its semantics. However, neglecting these invariants leads to performance issues. Singleton equalities enlarge needlessly the abstract states. Multiple equalities about

the same syntactic expression e make harder to find all expressions guaranteed to be equal to e by the abstract state. As these invariants are easy to maintain, we assume that they hold, and define the `find` operation accordingly.

Definition 63. We define a `find` operation of an expression e in an equality state D : it retrieves all the expressions that are guaranteed to evaluate to the same concrete value as e .

$$\text{find}(e, D) \triangleq \begin{cases} E & \text{if } \exists z \in \mathbb{Z}, (e, z) \in E \wedge E \in D \\ \emptyset & \text{otherwise} \end{cases}$$

The `find` operator allows us to define the forward and backward queries $F_{\mathbb{D}_{\text{eq}}}^{\#}$ and $B_{\mathbb{D}_{\text{eq}}}^{\#}$, by simply using the oracle or the reducer on the expressions that evaluate to the same concrete value as the requested expression. The forward query $F_{*_{\tau}}^{\#}$ on dereferences is implemented exactly as the query on arbitrary expressions $F_{\mathbb{D}_{\text{eq}}}^{\#}$, without using the location abstraction computed for the address. The backward query $B_{*_{\tau}}^{\#}$ on dereferences is the identity. Obviously, on an expression implying an equality or an inequality, the forward query of the domain may also answer directly, if the involved equality is known by the abstract state.

Definition 64 (Queries of the equality domain). Let \otimes be the concatenation of lists. Let D be an abstract state of the equality domain, e be an expression, l be a location abstraction, v be a value abstraction. The queries of the equality domain are defined as:

$$F_{\text{eq}}^{\#}(\text{oracle}, D, e) \triangleq \begin{cases} \top_v, \top_A & \text{if } \text{find}(e, D) = \emptyset \\ \bigcap_{(e', _) \in \text{find}(e, D)} (\text{fst}(\text{oracle}(e')), \perp_A) & \text{otherwise} \end{cases}$$

$$F_{*_{\tau}}^{\#}(\text{oracle}, D, e, l) \triangleq F_{\text{eq}}^{\#}(\text{oracle}, D, e)$$

$$F_{\mathbb{D}_{\text{eq}}}^{\#}(\text{oracle}, D, e) \triangleq \begin{cases} 1^{\#} & \text{if } e = (e_1 \diamond_{=} e_2) \wedge e_2 \in \text{find}(e_1, D) \\ 0^{\#} & \text{if } e = (e_1 \diamond_{\neq} e_2) \wedge e_2 \in \text{find}(e_1, D) \\ F_{\text{eq}}^{\#}(\text{oracle}, D, e) & \text{otherwise} \end{cases}$$

$$B_{*_{\tau}}^{\#}(D, l, v) \triangleq l$$

$$B_{\mathbb{D}_{\text{eq}}}^{\#}(D, e, v) \triangleq \bigotimes_{(e', _) \in \text{find}(e, D)} [(e', v)]$$

With $\diamond_{=} \in \{=, \leq, \geq\}$ and $\diamond_{\neq} \in \{\neq, <, >\}$.

Theorem 14. The queries of the equality domain defined above satisfy the requirements stated in definitions 56, 57 and 58.

Proof.

FORWARD QUERIES We prove here the soundness of $F_{\mathbb{D}_{\text{eq}}}^{\#}$; the proof for $F_{*_{\tau}}^{\#}$ is exactly the same. Let D be an abstract state of the equality domain, e be an expression and oracle be an oracle. Let S be a concrete state in $\gamma_{\text{eq}}(D) \cap \gamma_o(\text{oracle})$. We need to prove:

$$F_{\mathbb{D}_{\text{eq}}}^{\#}(\text{oracle}, D, e) \models_{\mathbb{V} \times \mathbb{A}} \llbracket e \rrbracket^{\Theta}(S)$$

The first two cases for $F_{\mathbb{D}_{\text{eq}}}^{\#}$ stem directly from the definition of the abstract states. As $S \in \gamma(D)$, we have:

$$\begin{aligned} e_2 \in \text{find}(e_1, D) &\Rightarrow \llbracket e_1 \rrbracket^{\Theta}(S) = \llbracket e_2 \rrbracket^{\Theta}(S) \\ &\Rightarrow \begin{cases} \llbracket e_1 \Diamond_{=} e_2 \rrbracket^{\Theta}(S) = 1 \\ \llbracket e_1 \Diamond_{\neq} e_2 \rrbracket^{\Theta}(S) = 0 \end{cases} \end{aligned}$$

We now consider the general case. Let $E = \text{find}(e, D)$. If E is empty, then $F_{\mathbb{D}_{\text{eq}}}^{\#}(\text{oracle}, D, e)$ is the top abstraction, which is sound by definition. Otherwise, $\exists z, (e, z) \in E$ and definition 62 of the concretization of the domain ensures that:

$$\exists V \in \mathcal{V}, \forall (e', z) \in E, \llbracket e' \rrbracket^{\Theta}(S) = V$$

This ensures that $\llbracket e \rrbracket^{\Theta}(S)$ succeeds, and thus $\perp_{\mathbb{A}} \models_{\mathbb{A}} \llbracket e \rrbracket^{\Theta}(S)$.

Moreover, for all $(e', z') \in E$, we have $\llbracket e' \rrbracket^{\Theta}(S) = \llbracket e \rrbracket^{\Theta}(S)$.

As $S \in \gamma_o(\text{oracle})$, the definition 54 of the oracle concretization ensures that:

$$\forall e' \in \text{expr}, \llbracket e' \rrbracket^{\Theta}(S) \in \gamma_{\mathbb{V}}(\text{fst}(\text{oracle}(e')))$$

Thus:

$$\forall (e', z') \in E, \llbracket e' \rrbracket^{\Theta}(S) \in \gamma_{\mathbb{V}}(\text{fst}(\text{oracle}(e')))$$

And by the soundness property of the meet operation,

$$\forall (e', z') \in E, \llbracket e \rrbracket^{\Theta}(S) \in \gamma_{\mathbb{V}}\left(\bigcap_{(e', z) \in \text{find}(e, D)} (\text{fst}(\text{oracle}(e')))\right)$$

Finally, we have

$$\left(\bigcap_{(e', z) \in \text{find}(e, D)} (\text{fst}(\text{oracle}(e'))), \perp_{\mathbb{A}}\right) \models_{\mathbb{V} \times \mathbb{A}} \llbracket e \rrbracket^{\Theta}(S)$$

QED.

BACKWARD QUERIES $B_{*_{\tau}}^{\#}$ being the identity function, its soundness is immediate. Let us prove the soundness of $B_{\mathbb{D}_{\text{eq}}}^{\#}$. Let D be an abstract state of the equality domain, let e be an expression and v

be a value abstraction. Let S be a concrete state in $\gamma_{\text{eq}}(S)$ such that $\llbracket e \rrbracket^\Theta(S) \in \gamma_v(v)$. We need to prove:

$$\begin{aligned} \forall(e', v') \in \mathbb{B}_{\mathbb{D}_{\text{eq}}}^\#(S, e, v), \llbracket e' \rrbracket^\Theta(S) \in \gamma_v(v') \\ \Leftrightarrow \forall(e, z) \in \text{find}(e, D), \llbracket e' \rrbracket^\Theta(S) \in \gamma_v(v) \end{aligned}$$

From the definition 63 of `find` and the definition 62 of the concretization of the equality domain, we know that:

$$\forall(e, z) \in \text{find}(e, D), \llbracket e' \rrbracket^\Theta(S) = \llbracket e \rrbracket^\Theta(S)$$

and we already have $\llbracket e \rrbracket^\Theta(S) \in \gamma_v(v)$. □

8.2.3 Interpretation of Assignments

On an assignment $*_\tau a := e$, the equality transformer needs to:

1. remove from the abstract state the equalities involving syntactic expressions whose value could be modified by the assignment;
2. computes an abstraction of the dependences of e and a .
3. adds the new equality $*_\tau a == e$ if it holds after the assignment, i.e. if the assignment does not modifies the value of a or e .

The second point requires some abstractions of the addresses dereferenced in expressions or assigned by assignment. As mentionned in Section 6.1.3, the valuations of EVA are made of a map from expressions to value abstractions, and a map from addresses to location abstractions. The latter map was omitted in the formalization of the evaluation, but it fulfills the same purpose and requirements as the expression map. The concretization of valuations is extended accordingly.

Definition 65 (Concretization of extended valuations). The concretization of valuation is extended to:

$$\begin{aligned} \gamma_{\mathbb{E}}(\mathcal{E}) \triangleq \{S \mid \forall e \in \text{dom}(\mathcal{E}), \llbracket e \rrbracket^\Theta(S) \in \gamma_v(\mathcal{E}(e)) \\ \wedge \forall a \in \text{dom}(\mathcal{E}), \llbracket a \rrbracket^\Theta(S) \in \gamma_{\mathbb{L}}(\mathcal{E}(a))\} \end{aligned}$$

Thus, the valuation makes available the location abstractions that the equality domain needs to interpret assignments.

8.2.3.1 Removing Equalities

We first define a `kill` function that removes from the equalities of an abstract state each expression whose dependence intersects a given memory location.

Definition 66. Let l be a location abstraction. We define the function kill as:

$$\begin{aligned} \text{kill}_{\mathbb{E}} : \mathbb{Z}^{\#} &\rightarrow \mathbb{E} \rightarrow \mathbb{E} \\ \text{kill}_{\text{eq}} : \mathbb{L}^{\#} &\rightarrow \text{type} \rightarrow \mathbb{D}_{\text{eq}} \rightarrow \mathbb{D}_{\text{eq}} \end{aligned}$$

$$\begin{aligned} \text{kill}_{\mathbb{E}}(z, E) &\triangleq \{(e, z') \mid (e, z') \in E \wedge z \sqcap_{\mathbb{Z}} z' = \perp\} \\ \text{kill}_{\text{eq}}(l, \tau, D) &\triangleq \{\text{kill}_{\mathbb{E}}(\text{zone}(l, \tau), E) \mid E \in D\} \end{aligned}$$

Lemma 24. Let $*_{\tau}a := e$ be an assignment, l a location abstraction and D an abstract state of the equality domain. If D is a sound approximation of the concrete states before the statement and if l is a sound approximation of the address a in these concrete states, then $\text{kill}_{\text{eq}}(l, \tau, D)$ is a sound approximation of the concrete states after the execution of the assignment.

$$\forall S \in \gamma_{\text{eq}}(D), \overline{[a]}^{\Theta}(S) \in \gamma_{\mathbb{L}}(l) \Rightarrow \llbracket *_{\tau} a := e \rrbracket(S) \in \gamma_{\text{eq}}(\text{kill}_{\text{eq}}(l, \tau, D))$$

Proof. Let $*_{\tau}a := e$ be an assignment and let l be a sound approximation of the value of a . Let D be an abstract state of the equality domain and let $D' = \text{kill}_{\text{eq}}(l, \tau, D)$. Let S a concrete state in $\gamma_{\text{eq}}(D)$ for which the interpretation of the assignment succeeds, and let $S' = \llbracket *_{\tau} a := e \rrbracket(S)$. We want to prove that $S' \in \gamma_{\text{eq}}(D')$. This requires that:

$$\forall E' \in D', \exists V \in \mathcal{V}, \forall (e, z) \in E', \begin{cases} \overline{[e]}^{\Theta}(S') = V \\ \text{deps}(e, S') \in \gamma_{\mathbb{Z}}(z) \end{cases}$$

Let E' be an equality of D' . By the definition of kill , we have:

$$\begin{aligned} \exists E \in D, E' &= \text{kill}_{\mathbb{E}}(\text{zone}(l, \tau), E) \\ \Leftrightarrow \exists E \in D, E' &= \{(e, z) \mid (e, z) \in E \wedge z \sqcap_{\mathbb{Z}} \text{zone}(l, \tau) = \perp\} \end{aligned} \quad (8.2)$$

As $E \in D$ and $S \in \gamma_{\text{eq}}(D)$, definition 62 states that it exists a concrete value V such that:

$$\forall (e, z) \in E, \begin{cases} \overline{[e]}^{\Theta}(S) = V \\ \text{deps}(e, S) \in \gamma_{\mathbb{Z}}(z) \end{cases} \quad (8.3)$$

We also have $S' = \llbracket *_{\tau} a := e \rrbracket(S)$.

Let $L \in \mathcal{L}_{\tau}$ such that $\forall \theta \in \Theta_P, \overline{[a]}_{S(\theta)} = L$. Such a concrete location exists, as the execution of the assignment succeeds in S , and we have $\lambda_{\cdot}.L \in \gamma_{\mathbb{L}}(l)$, since l is a sound approximation of the address a . The state S' can then be rewritten:

$$S' = \lambda \theta. S(\theta)[\text{loc}_{\tau}(L) \mapsto \phi_{\tau}^{-1}(V(\theta))]$$

This implies that:

$$\forall \theta \in \Theta_P, \forall l \notin \text{loc}_{\tau}(L), S(\theta)(l) = S'(\theta)(l) \quad (8.4)$$

Now, let $(e, z) \in E'$. By 8.2:

$$(e, z) \in E \wedge z \sqcap_{\mathbb{Z}} \text{zone}(l, \tau) = \perp \quad (8.5)$$

By the concretization 62 of the equality domain: $\text{deps}(e, S) \subseteq \gamma_{\mathbb{Z}}(z)$

As $\lambda_{\perp}.L \in \gamma_{\mathbb{L}}(l)$, by definition 61: $\text{loc}_{\tau}(L) \subseteq \gamma_{\mathbb{Z}}(\text{zone}(l, \tau))$

As $z \sqcap_{\mathbb{Z}} \text{zone}(l, \tau) = \perp$, we have $\gamma_{\mathbb{Z}}(z) \cap \gamma_{\mathbb{Z}}(\text{zone}(l, \tau)) \subseteq \gamma_{\mathbb{Z}}(\perp_{\mathbb{Z}}) = \emptyset$

And thus $\text{deps}(e, S) \cap \text{loc}_{\tau}(l) \subseteq \gamma_{\mathbb{Z}}(z) \cap \gamma_{\mathbb{Z}}(\text{zone}(l, \tau)) = \emptyset$

And by 8.4 and 8.5:

$$(e, z) \in E \wedge \forall \theta \in \Theta_P, \forall l \in \text{deps}(e, S), S(\theta)(l) = S'(\theta)(l) \quad (8.6)$$

By equation 8.3:

$$\begin{cases} \llbracket e \rrbracket^{\Theta}(S) = V \\ \text{deps}(e, S) \in \gamma_{\mathbb{Z}}(z) \end{cases}$$

Finally, lemma 23 and equation 8.6 ensure that:

$$\llbracket e \rrbracket^{\Theta}(S) = \llbracket e \rrbracket^{\Theta}(S') \wedge \text{deps}(e, S) = \text{deps}(e, S')$$

This means that

$$\forall (e, z) \in E', \begin{cases} \llbracket e \rrbracket^{\Theta}(S') = V \\ \text{deps}(e, S') \in \gamma_{\mathbb{Z}}(z) \end{cases}$$

QED. □

8.2.3.2 Computing Dependences

We now present the computation of an abstraction of the dependences of expressions in the concrete states represented by a valuation.

Definition 67 (Computation of dependences abstractions). We define $\text{deps}^{\#} : \text{expr} \rightarrow \mathbb{E} \rightarrow \mathbb{Z}^{\#}$ inductively as:

$$\begin{aligned} \text{deps}^{\#}(*_{\tau}a, \mathcal{E}) &\triangleq \text{zone}(\mathcal{E}(a), \tau) \sqcup_{\mathbb{Z}} \text{deps}^{\#}(a, \mathcal{E}) \\ \text{deps}^{\#}(\diamond(e_1, \dots, e_n), \mathcal{E}) &\triangleq \sqcup_{i \in \{1, \dots, n\}} \text{deps}^{\#}(e_i, \mathcal{E}) \\ \text{deps}^{\#}(cst, \mathcal{E}) &\triangleq \perp_{\mathbb{Z}} \\ \text{deps}^{\#}(t[i], \mathcal{E}) &\triangleq \text{deps}^{\#}(t, \mathcal{E}) \sqcup_{\mathbb{Z}} \text{deps}^{\#}(i, \mathcal{E}) \\ \text{deps}^{\#}(e.\text{field}, \mathcal{E}) &\triangleq \text{deps}^{\#}(e, \mathcal{E}) \end{aligned}$$

Lemma 25. Let \mathcal{E} be a valuation, e be an expression and a be an address. The zone abstractions $\text{deps}^{\#}(e, \mathcal{E})$ and $\text{deps}^{\#}(a, \mathcal{E})$ are respectively sound approximations of $\text{deps}(e, S)$ and $\text{deps}(a, S)$ for all states S in $\gamma_{\mathbb{E}}(\mathcal{E})$.

$$\forall \mathcal{E} \in \mathbb{E}, \forall e \in \text{expr}, \forall a \in \text{addr},$$

$$\forall S \in \gamma_{\mathbb{E}}(\mathcal{E}), \begin{cases} \text{deps}(e, S) \subseteq \gamma_{\mathbb{Z}}(\text{deps}^{\#}(e, \mathcal{E})) \\ \text{deps}(a, S) \subseteq \gamma_{\mathbb{Z}}(\text{deps}^{\#}(a, \mathcal{E})) \end{cases}$$

Proof. By induction on the expression e and the address a , using that the join $\sqcup_{\mathbb{Z}}$ is an overapproximation of the union of sets \cup (dependences are sets of bytes). The base case, for constants cst , is trivial: $\gamma_{\mathbb{Z}}(\perp_{\mathbb{Z}}) = \emptyset$ by definition. We prove the case of dereferences, assuming the induction hypothesis. Let \mathcal{E} be a valuation, τ a scalar type and a an address. Let S be a concrete state in $\gamma_{\mathbb{E}}(\mathcal{E})$.

$$\text{deps}(a, S) \subseteq \gamma_{\mathbb{Z}}(\text{deps}^{\#}(a, \mathcal{E})) \Rightarrow \text{deps}(*_{\tau}a, S) \subseteq \gamma_{\mathbb{Z}}(\text{deps}^{\#}(*_{\tau}a, \mathcal{E}))$$

We recall the definition of the concrete and abstract dependences of the dereference:

$$\begin{aligned} \text{deps}(*_{\tau}a, S) &= \text{loc}_{\tau}(L) \cup \text{deps}(a, S) \\ &\quad \text{where } \forall \theta \in \Theta_P, \llbracket a \rrbracket_{S(\theta)} = L \in \mathcal{L}_{\tau} \\ \text{deps}^{\#}(*_{\tau}a, \mathcal{E}) &= \text{zone}(\mathcal{E}(a), \tau) \sqcup_{\mathbb{Z}} \text{deps}^{\#}(a, \mathcal{E}) \end{aligned}$$

We have $S \in \gamma_{\mathbb{E}}(\mathcal{E})$ and $\llbracket a \rrbracket^{\Theta}(S) = \lambda_{-}.L$

By definition 65: $(\lambda_{-}.L) \in \gamma_{\mathbb{L}}(\mathcal{E}(a))$

By definition 61: $\text{loc}_{\tau}(L) \subseteq \gamma_{\mathbb{Z}}(\text{zone}(l, \tau))$

By induction hypothesis: $\text{deps}(a, \mathcal{E}) \subseteq \gamma_{\mathbb{Z}}(\text{deps}^{\#}(a, \mathcal{E}))$

By these equations and the soundness of the join:

$$\begin{aligned} \text{loc}_{\tau}(L) \cup \text{deps}(a, S) &\subseteq \gamma_{\mathbb{Z}}(\text{zone}(l, \tau)) \cup \gamma_{\mathbb{Z}}(\text{deps}^{\#}(a, \mathcal{E})) \\ &\subseteq \gamma_{\mathbb{Z}}(\text{zone}(\mathcal{E}(a), \tau) \sqcup_{\mathbb{Z}} \text{deps}^{\#}(a, \mathcal{E})) \end{aligned}$$

QED. □

8.2.3.3 Adding New Equalities

After an assignment $*_{\tau}a := e$, the equality between $*_{\tau}a$ and e holds, unless if the assignment modifies the value of the expression e or of the value of the address a .

Example 31. An example of the first case is the assignment $*_{\tau}(\&x, 0) = *_{\tau}(\&x, 0) +_{\tau} 1$ where τ is a scalar type. This assignment is written $x = x+1$ in the C syntax. Obviously, the equality $x = x + 1$ does not hold after the assignment.

Example 32. Figure 8.1 presents two code snippets where the execution of the final assignment $*_{\tau}a := e$ modifies the value of the syntactic address a . At line 2 on the left code, $t[t[0]]$ refers to $t[0]$, the first cell of the array t , which is thus modified by the assignment. Afterwards, $t[0]$ contains the value 2 and $t[t[0]]$ refers to $t[2]$, the third cell of the array. The equality $t[t[0]] == 2$ is false, as $t[2]$ has value 0.

At line 3 on the right code, $*(p+i)$ refers to $\&i$, and the assignment modifies the value of the variable i . Afterwards, the pointer $*(p+i)$ refers to $\&(i+1)$, which is not even a valid address to read at: the equality $*(p+i) == 1$ does not hold.

```
int t[5] = {0; 0; 0; 0; 0}
int t[t[0]] = 2;
```

```
int i = 0;
int *p = &i;
*(p+i) = 1;
```

Figure 8.1: Assignment modifying the address

Lemma 26. *Let $*_{\tau}a := e$ be an assignment. Let S be a concrete state in which the execution of the assignment succeeds. Let a location abstraction l be a sound approximation of the value of a in S . Let two zone abstractions z_a and z_e be sound approximations of the dependences of respectively a and e in S . If $\text{zone}(l, \tau) \sqcap_{\mathbb{Z}} z_a$ and $\text{zone}(l, \tau) \sqcap_{\mathbb{Z}} z_e$ are bottom, then the syntactic equality $*_{\tau}a = e$ holds after the assignment, i.e. in the state $\llbracket *_{\tau} a := e \rrbracket(S)$. Moreover, the zone abstractions z_a and z_e are still sound approximations of the dependences of e and a in this post state.*

*If we write $S' = \llbracket *_{\tau} a := e \rrbracket(S)$:*

$$\left. \begin{array}{l} \llbracket a \rrbracket^{\Theta}(S) \in \gamma_{\mathbb{L}}(l) \\ \text{deps}(a, S) \subseteq z_a \\ \text{deps}(e, S) \subseteq z_e \\ \text{zone}(l, \tau) \sqcap_{\mathbb{Z}} z_e = \perp_{\mathbb{Z}} \\ \text{zone}(l, \tau) \sqcap_{\mathbb{Z}} z_a = \perp_{\mathbb{Z}} \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \llbracket *_{\tau} a \rrbracket^{\Theta}(S') = \llbracket e \rrbracket^{\Theta}(S') \\ \text{deps}(a, S') \subseteq z_a \\ \text{deps}(e, S') \subseteq z_e \end{array} \right.$$

Proof. This proof uses the same notations as the lemma. As the execution of the assignment succeeds in the concrete state S , $\llbracket a \rrbracket^{\Theta}(S)$ is a constant location (independent of the memory location). We then write:

$$(\lambda_{-}.L) = \llbracket a \rrbracket^{\Theta}(S) \quad V = \llbracket e \rrbracket^{\Theta}(S) \quad S' = \llbracket *_{\tau} a := e \rrbracket(S)$$

As l is a sound approximation of the value of a in S : $(\lambda_{-}.L) \in \gamma_{\mathbb{L}}(l)$

By definition 61, $\text{loc}_{\tau}(L) \subseteq \text{zone}(l, \tau)$

By soundness of zones: $\text{deps}(a, S) \subseteq \gamma_{\mathbb{Z}}(z_a) \wedge \text{deps}(e, S) \subseteq \gamma_{\mathbb{Z}}(z_e)$

By the soundness property of the meet:

$$\begin{aligned} \text{zone}(l, \tau) \sqcap_{\mathbb{Z}} z_a = \perp_{\mathbb{Z}} &\Rightarrow \text{loc}_{\tau}(L) \cap \text{deps}(a, S) = \emptyset \\ \text{zone}(l, \tau) \sqcap_{\mathbb{Z}} z_e = \perp_{\mathbb{Z}} &\Rightarrow \text{loc}_{\tau}(L) \cap \text{deps}(e, S) = \emptyset \end{aligned}$$

Thanks to the concrete semantics of assignment (Figure 4.9), the post state S' can be written as:

$$S' = \lambda\theta. S(\theta)[\text{loc}_{\tau}(L) \mapsto \phi_{\tau}^{-1}(V(\theta))] \quad (8.7)$$

The concrete states S and S' coincide on all bytes but $\text{loc}_{\tau}(L)$. As the intersection between $\text{loc}_{\tau}(L)$ and the dependences of e and a in S is empty, lemma 23 applies, and we have:

$$\begin{aligned} \llbracket e \rrbracket^{\Theta}(S') &= \llbracket e \rrbracket^{\Theta}(S) \wedge \text{deps}(e, S) = \text{deps}(e, S') \\ \llbracket a \rrbracket^{\Theta}(S') &= \llbracket a \rrbracket^{\Theta}(S) \wedge \text{deps}(a, S) = \text{deps}(a, S') \end{aligned}$$

This already ensures that $\text{deps}(a, S') \subseteq \gamma_{\mathbb{Z}}(z_a) \wedge \text{deps}(e, S') \subseteq \gamma_{\mathbb{Z}}(z_e)$. The zone abstractions z_a and z_e are still sound approximations of the dependences of a and e in the post state S' .

Moreover, we have:

$$\begin{aligned}\overline{\llbracket e \rrbracket}^{\Theta}(S') &= \overline{\llbracket e \rrbracket}^{\Theta}(S) = V \\ \overline{\llbracket a \rrbracket}^{\Theta}(S') &= \overline{\llbracket a \rrbracket}^{\Theta}(S) = \lambda_{\perp}.L\end{aligned}$$

By the concrete semantics of expressions (Figure 4.8) and 8.7:

$$\begin{aligned}\overline{\llbracket *_{\tau} a \rrbracket}^{\Theta}(S) &= \lambda\theta. (S'(\theta))_{\tau}[l] \\ &= \lambda\theta. \phi'_{\tau}(S'(\theta)(\text{loc}_{\tau}(L))) \\ &= \lambda\theta. \phi'_{\tau}(\phi_{\tau}^{-1}(V(\theta))) \\ &= \lambda\theta. V(\theta) \\ &= V = \overline{\llbracket e \rrbracket}^{\Theta}(S')\end{aligned}$$

The equality $*_{\tau} a = e$ holds in the post state S' . \square

The last requirement to finally define the transfer function on assignments is the addition of a new equality into an abstract state. We assume given an operator \oplus_{eq} that adds a new equality between two expressions (and their dependences) to an abstract state. It must satisfy the following lemma.

Proposition 9. *Let D be an abstract state of the equality domain, e_1 and e_2 be two expressions, and z_1 and z_2 be two zone abstractions. We write $D' = \{(e_1, z_1), (e_2, z_2)\} \oplus_{\text{eq}} D$. Then:*

$$\forall S \in \gamma_{\text{eq}}(D), \left. \begin{array}{l} \overline{\llbracket e_1 \rrbracket}^{\Theta}(S) = \overline{\llbracket e_2 \rrbracket}^{\Theta}(S) \\ \text{deps}(e_1, S) \subseteq z_1 \\ \text{deps}(e_2, S) \subseteq z_2 \end{array} \right\} \Rightarrow S \in \gamma_{\text{eq}}(D')$$

8.2.3.4 Complete Interpretation of Assignments

Definition 68. Let $*_{\tau} a := e$ be an assignment, D be an abstract state of the equality domain, \mathcal{E} be a valuation containing an abstraction for each subterm of a and e .

$$\begin{aligned}T_{\text{eq}}(*_{\tau} a := e, \mathcal{E}, D) &\triangleq \begin{cases} E' \oplus_{\text{eq}} D' & \text{if } z_a \sqcap_{\mathbb{Z}} z = z_e \sqcap_{\mathbb{Z}} z = \perp_{\mathbb{Z}} \\ D' & \text{otherwise} \end{cases} \\ \text{where } \begin{cases} z_a = \text{deps}^{\#}(a, \mathcal{E}) \\ z_e = \text{deps}^{\#}(e, \mathcal{E}) \\ z = \text{zone}(\mathcal{E}(a), \tau) \\ E' = \{(*_{\tau} a, z_a), (e, z_e)\} \\ D' = \text{kill}(\mathcal{E}(a), D) \end{cases} \end{aligned}$$

Theorem 15. Let $*_{\tau}a := e$ be an assignment, D be an abstract state of the equality domain, \mathcal{E} be a valuation. For all the concrete states S in $\gamma_{\text{eq}}^{\mathbb{E}}(D, \mathcal{E})$ for which the execution of the statement succeeds, the concrete state $\overline{\llbracket *_{\tau}a := e \rrbracket}(S)$ is in $\gamma_{\text{eq}}(\mathbf{T}_{\text{eq}}(*_{\tau}a := e, \mathcal{E}, D))$.

In other words, \mathbf{T}_{eq} satisfies the property of definition 59, and is thus a sound abstract transformer of assignments.

Proof. Let $*_{\tau}a := e$ be an assignment, D be an abstract state of the equality domain, \mathcal{E} be a valuation containing an abstraction for each subterm of a and e .

Let S be a concrete state in $\gamma_{\text{eq}}^{\mathbb{E}}(D, \mathcal{E})$. We write $S' = \overline{\llbracket *_{\tau}a := e \rrbracket}(S)$, $D'' = \mathbf{T}_{\text{eq}}(*_{\tau}a := e, \mathcal{E}, D)$ and want to prove that $S' \in \gamma_{\text{eq}}(D'')$.

As $S \in \gamma_{\mathbb{E}}(\mathcal{E})$, by definition 65: $\overline{\llbracket a \rrbracket}^{\Theta}(S) \in \gamma_{\mathbb{L}}(\mathcal{E}(a))$.

By lemma 24: $S' \in \gamma_{\text{eq}}(\text{kill}_{\text{eq}}(\mathcal{E}(a), \tau, D))$

In the second case of the definition, $D'' = \text{kill}_{\text{eq}}(\mathcal{E}(a), \tau, D)$, and the proof is completed. We now consider the first case only.

Lemma 25 ensures that the zones $z_a = \text{deps}^{\#}(a, \mathcal{E})$ and $z_e = \text{deps}^{\#}(e, \mathcal{E})$ are sound approximations of the dependences of a and e in the concrete states of $\gamma_{\mathbb{E}}(\mathcal{E})$:

$$\begin{cases} \text{deps}(e, S) \subseteq \gamma_{\mathbb{Z}}(z_e) \\ \text{deps}(a, S) \subseteq \gamma_{\mathbb{Z}}(z_a) \end{cases}$$

And we already know $\overline{\llbracket a \rrbracket}^{\Theta}(S) \in \gamma_{\mathbb{L}}(\mathcal{E}(a))$.

Thus, lemma 26 ensures that:

$$\left. \begin{array}{l} \text{zone}(\mathcal{E}(a), \tau) \sqcap_{\mathbb{Z}} z_e = \perp_{\mathbb{Z}} \\ \text{zone}(\mathcal{E}(a), \tau) \sqcap_{\mathbb{Z}} z_a = \perp_{\mathbb{Z}} \end{array} \right\} \Rightarrow \begin{cases} \overline{\llbracket *_{\tau}a \rrbracket}^{\Theta}(S') = \overline{\llbracket e \rrbracket}^{\Theta}(S') \\ \text{deps}(e, S') \subseteq \gamma_{\mathbb{Z}}(z_e) \\ \text{deps}(a, S') \subseteq \gamma_{\mathbb{Z}}(z_a) \end{cases}$$

The new equality holds in the concrete state S' , and the dependences are still correct. Proposition 9 ensures that the new abstract state $\{(*_{\tau}a, z_a), (e, z_e)\} \oplus_{\text{eq}} S'$ is a sound abstraction of S . In all cases:

$$S' \in \gamma_{\text{eq}}(D'')$$

□

8.2.4 Interpretation of Other Statements

The interpretation of other statements than assignments is straightforward.

On a test filter $(e_1 \neq e_2) == 0?$, the equality $e_1 = e_2$ is added to the abstract state. In the C syntax, this corresponds to the *if* branch of `if(e1==e2)` or the *else* branch of `if(e1!=e2)`. Otherwise, the equality domain cannot learn anything from a test filter, and the abstract state after the statement is the same as the abstract state before.

Finally, the abstract transformer for the entry in scope of variables is the identity; the abstract transformer for the exit of scope of variables removes from the state the expressions that depends on these variables.

8.2.5 Implementation

An equality is encoded as a set of at least two expressions. A set of equalities is encoded as a map from expressions to equalities: it binds each involved expression to the equality containing it. This representation makes efficient the `find` operation, used for every query during the evaluation. On the other hand, the addition or removal of an equality can be costly, as the modification of an existing equality E requires to update the binding of each expression in E . In an abstract state where e_1 and e_2 belong respectively to the disjoint equalities E_1 and E_2 , adding the equality $e_1 = e_2$ amounts to merge E_1 and E_2 , and requires $|E_1| + |E_2|$ binding updates. In an abstract state where e belongs to E , removing e —when e can be modified by an assignment—requires $|E|$ binding updates. In practice, the cardinal of the equalities propagated by the domain remains small enough to not overly impeded the analysis.

Two last optimizations limit the number of equalities created by the domain. Firstly, if the removal of e makes E a singleton, then E is totally removed instead. Secondly, the addition of a new equality $e_1 = e_2$ is avoided if the two abstractions for e_1 and e_2 in the valuation are singleton—if they represent only one concrete value. (For the equality stemming from the assignment $*_r a := e$, this means that the abstractions for a and e are singleton.) In this case, the equality is *de facto* already known by the abstract states of the other domains.

The join of two abstract states must keep only the equalities carried by both states. It is simply a merge of both maps, by intersecting pointwise the equalities bound to the same expressions. Expressions belonging to only one map are simply removed. The maps used by the domain are patricia trees [OG98], where the placement of a data is fixed by the bits of its key. This allows efficient merges, by traversing the two maps at the same time.

As shown in the previous sections, the abstract domain relies on dependence abstractions to remove from the states the expressions whose value may be modified by assignments. These dependence abstractions are built upon the location abstractions cooperatively computed by the evaluation, and made available by the valuation. To efficiently retrieve the expressions to be removed at an assignment, the domain also maintains a dependency table—implemented through the generic datastructures of `CVALUE`—that links a location to the expressions that depends on it.

	loc	statements	EVA	EVA+EQ	
idct	342	586 (94%)	1.4s	1.7	time
			56	56	alarms
tweetnacl	799	658 (99%)	12s	15s	time
			2	2	alarms
papabench	3199	1845 (55%)	1.3s	1.4s	time
			31	31	alarms
debic	5326	2872 (98%)	13s	13s	time
			55	42	alarms
gzip	5196	4170 (94%)	42s	53s	time
			1505	1501	alarms
polarssl	24k	2758 (49%)	230s	294s	time
			158	149	alarms
indus1	99k	30955 (81%)	430s	686s	time
			228	222	alarms
indus2	125k	36782 (80%)	715s	1343s	time
			244	238	alarms

Table 8.3: Experimental results of the equality domain

Finally, the domain only relies on the oracle and the reducer to avail the relations it infers. It is thus independent of the chosen value abstraction, and is implemented as a functor from value to state abstraction.

8.2.6 Experimental Results

In practice, the equality domain of EVA behaves exactly as explained in examples 26 and 29.

Table 8.3 presents the experimental results obtained with the equality domain on the programs already used in Section 8.1.3. It compares the analysis time and the number of emitted alarms by EVA’s analysis instantiated with only the CVALUE domain, or with the product of the CVALUE domain and the equality domain. In 5 out of 8 programs (all those with a significant number of alarms), less alarms are emitted. The slowdown introduced by the equality domain does not exceed 30%, except on the two industrial case studies, on which the analysis is 60% and 88% slower. Still, gaining 6 alarms there (on analyses that use loop unrolling and disjunctive completion very aggressively for maximum precision) is already a very positive result, given the limited relationality of the equality domain.

8.3 OTHER NEW DOMAINS IN EVA

In addition to the equality domain, four other new abstract domains have been introduced in [EVA](#) during the last year: a binding to the numerical abstract domains provided by the [APRON](#) library, a domain of symbolic locations, a gauges domain and a bitwise domain. The first three domains are purely numerical. They process the assignments through pointers in the same way as the equality domain: they rely on the location abstractions computed by the [CVALUE](#) domain to remove the properties invalidated by the assignment. The bitwise domain is built upon the same memory model than the [CVALUE](#) domain, reusing the generic OCaml functors of its data structures.

This section briefly describes these new domains, and presents their performances on the same case studies used before in this chapter. It should be remembered that the implementation of these domains remain mostly experimental.

8.3.1 *Binding to the APRON domains*

We have implemented a minimal binding to the numerical domains available in [APRON](#) [[JMo9](#)]. The resulting domains (boxes, octagons, strict or loose convex polyhedra, linear equalities) are proofs of concept to demonstrate that preexisting relational domains fit within the communication model of [EVA](#), and are easy to introduce. Our [APRON](#) binding has currently less than 1000 lines of code. The abstract state is an [APRON](#) state. A (static) mapping between the [APRON](#) dimensions and the scalar variables present in the program is created by the binding, and used as a *correspondence* table to translate the memory locations inferred by the [CVALUE](#) domain. Indeed, the [APRON](#) domains bring no information about which variable a pointer may point to; instead, they entirely rely on the other domains for aliasing information. The domain answers queries for arithmetic expressions, that are translated into the [APRON](#) internal language. Sub-expressions that cannot be handled by [APRON](#) are linearized on-the-fly into intervals, using the cooperatively computed value. Currently, the binding is limited to variables with integer types.

We plan to add support for variables with floating-point types. Tracking information for structs or arrays is also easy, and simply requires extending the correspondence table. However, the most important improvement would consist in using *variable packing* [[Cou+09](#)]. Analyzing large problems in which all variables are in relation is known to be unfeasible, due to the high algorithmic complexity of relational domains. The domain should instead infer subsets of “related” variables, and only track relations within those sets. The difficulty lies in finding good heuristics, which should then be implemented on the [EVA](#) side of the domain.

8.3.2 The Symbolic Locations Domain

The symbolic locations domain tracks accesses to arrays or through pointers in a symbolic way. Its intent is to analyze code such as `if (t[i] <= e) v = t[i];` in a precise way. Indeed, when `i` is imprecise, domains that represent arrays in extenso cannot learn information from the condition (because any cell may be involved). The domain shares some similarities with the recency abstraction [BRo6]. Its state is a map from symbolic locations (such as `t[i]`, `*p` or `p->v`) to an abstract value. Strong reductions may be performed on those values when analyzing conditions, to be shared with the other domains when the location is encountered again later.

8.3.3 The Gauges Domain

Gauges [Ven12] are a weakly relational domain, able to efficiently infer general linear inequality invariants within loops. Technically, the variables involved in the invariants are all related to loop counters, that model the current number of iterations in each loop. Gauges are especially useful to infer invariants for pointer offsets, as pointer arithmetic introduces `+4` or `+8` increments (for 32- and 64-bits architecture respectively), that cannot be directly handled by domains such as octagons. The gauges domain communicates integer and pointer values through the standard values of EVA.

8.3.4 Bitwise Abstractions

The bitwise domain aims at adding bitvector-like reasoning to EVA (including on floating-point values and pointers), without resorting to a dedicated implementation. Instead, we reuse the expressivity of the offsetmaps, that represent sequences of bits in the cvalue domain. Indeed, this abstraction is already able to extract the possible values of some bits in a memory range. This bitwise domain works on a new kind of value abstractions, namely a sequence of bits of known length. Only the forward and backward abstract semantics for the bitwise C operators, as well as integer casts and multiplication/division by a power of 2, have been implemented. All other operations degenerate to \top_v . The reduced product between the standard values and those new bitwise values performs a conversion between the two representations when possible.

8.3.5 Experimental Results

Figure 8.2 presents the results of some experiments conducted on the new abstract domains of our analyzer. We compare the precision and the analysis time of:

	loc	EVA	Eqs	Loc	Gau	All	
tweetnacl	799	12s	15s	14s	15s	19s	time
		2	2	2	2	2	alarms
papabench	3199	1.3s	1.4s	1.3s	1.3s	1.6s	time
		31	31	31	31	31	alarms
debis	5326	11s	13s	14s	13s	16s	time
		55	42	55	53	40	alarms
gzip	5196	42s	53s	—	60s	—	time
		1505	1501	—	1500	—	alarms
polarssl	24k	230s	294s	266s	382s	432s	time
		158	149	156	152	140	alarms
indus	125k	850s	1090s	1390s	1120s	1930s	time
		153	136	150	153	134	alarms

Figure 8.2: Experimental results on the new abstract domains

- EVA with the default CVALUE domain (EVA);
- EVA with the CVALUE domain, augmented respectively with:
 - the equality domain (Eqs);
 - the symbolic locations domain (Loc);
 - the gauges domain (Gau);
 - all these three domains (All).

We observe some improvements on two case studies, debis and polarssl, where the introduction of the new domains allows the analyzer to rule out a few more alarms. The slowdown introduced by each domain basically ranges between 15 to 66% from case to case. On polarssl, the largest open-source case study, the analysis with all these domains is twice as slow than the analysis with the CVALUE domain only, and proves 11% of the alarms emitted by the CVALUE domain.

These results are quite encouraging, given that the new domains remain experimental and mostly unoptimized. However, much work remains to be done to bring the new domain to the same level of maturity than the CVALUE domain.

8.3.6 Conclusion

This concludes our presentation of the EVA analyzer. The alarms and the value abstractions used by the analyzer have been formalized in Chapter 5. The strategies for the cooperative computation

of these abstractions of expressions has been detailed in Chapter 6. Chapter 7 illustrated how this cooperative computation can assist abstract domains to precisely interpret the statement semantics. Finally, this chapter described the new abstract domains that have been implemented within EVA, and how they fit into its architecture. Even though the new abstractions are not as efficient as the legacy CVALUE domain yet, they fit properly in the architecture of our analyzer. In our experience, the communication through value abstractions does not restrain the expressivity of the product of abstract domains, and the cooperative evaluation of expressions effectively assists the implementation of new domains. We believe that the variety of abstractions introduced within EVA in less than a year validates its core concepts and its design.

Part V

ABSTRACT SEMANTICS OF TRACES

This chapter differs slightly from the other parts of this thesis. While the preceding chapters are mostly dedicated to the structuring of a hierarchy of abstractions, and to the organization of interactions between them, this chapter presents a generic framework to improve the precision of any abstract domain at join points. Moreover, this work has not been implemented in [EVA](#), but as a standalone new plugin of FRAMA-C.

Abstract semantics often perform wide approximations when two control-flow paths meet, by merging the states from each path. This join of abstract states ensures the soundness of the abstract semantics, but is also a common source of imprecision: the properties inferred in only one path are generally lost. This chapter presents how to mechanically augment any standard domain with conditional predicates to circumvent this imprecision. The predicates are derived from conditional statements, and postpone the loss of information. The resulting domain is called a predicated domain, and analyses over such a domain are called predicated analyses. This work has been published in [\[BBY16\]](#).

After detailing our motivations, this chapter formalizes the semantics of a predicated domain, and explains how to build an efficient predicated analysis. It also outlines the coq proof of the soundness of a predicated analysis, exposes some related works, and describes the experimental evaluation of our practical implementation.

9.1 MOTIVATION

Seeking to strike the best balance between precision and efficiency is always a challenge in abstract interpretation. *Flow-sensitivity*, which allows to infer static properties that depend on program points, is often considered as a prerequisite to obtain a precise program analysis. More aggressive analyses are *path-sensitive*: the analysis of a program statement depends on the control-flow path followed to reach this statement. Nevertheless, most analyses sacrifice full path-sensitivity and perform approximations when two control-flow paths meet. Those approximations may lead to a significant loss of precision, and may preclude inferring some interesting properties of the program.

Consider as an example the code fragment of [Figure 9.1](#), which is a simplified version of a real-life program that opens and closes file descriptors. Proving that the three calls to the `close` function

```

1  if (flag1)
    { fd1 = open(path1);
      if (fd1 == -1) exit(); }
  [...] // code 1
5  if (flag2)
    { fd2 = open(path2);
      if (fd2 == -1) {
        if (flag1) close(fd1);
        exit(); } }
10 [...] // code 2
    if (flag1) close(fd1);
    if (flag2) close(fd2);

```

Figure 9.1: Example of interleaved conditionals

are correct, i.e. that the corresponding `fd` variable has been properly created following the calls to the `open` function, heavily relies on the possible values for the `flag1` and `flag2` variables. An analysis that does not keep track of the relation between `flag1` and `fd1` on the one hand, and `flag2` and `fd2` on the other hand, will not be able to prove that the program is correct.

In this chapter, we define an analysis in which information about the conditionals that have been encountered so far is retained using boolean predicates. These predicates guard the values inferred about the program. Our analysis is parameterized by a pre-existing analysis domain, which we use to derive a new *predicated* analysis. More precisely, we propagate two kinds of information that are not present in the original domain: a context and an implication map.

1. A *context* is a boolean predicate synthesized from the guards of the conditionals that have been reached so far, and that is guaranteed to hold at the current program point. In our example, at the beginning of line 8, the context would be $\text{flag2} \wedge (\text{fd2} = -1)$.
2. An *implication map* is a set of facts from the original analysis domain, guarded by boolean predicates. Each fact is guaranteed to hold when its guard holds. In this example, we suppose that the analysis domain keeps track of whether `open` returned a valid file descriptor, or `-1` in case of error. Here are the implications we would like to infer after line 6:

$$\text{flag1} \mapsto \text{valid_fd}(\text{fd1}) \quad \text{true} \mapsto \text{valid_fd}(\text{fd2}) \vee (\text{fd2} = -1)$$

The first implication results from the analysis of the conditionals at lines 1–3; it precisely models the information we need between `flag1` and `fd1`. The second implication is simply the postcondition of the `open` function, which holds unconditionally.

$$\begin{array}{ll}
e \in \text{expr} & ::= \quad x \quad \quad x \in \mathcal{X} \\
& \quad | \quad v \quad \quad v \in \overline{\mathbb{V}} \\
& \quad | \quad e \diamond e \\
c, p \in \mathbb{C} & ::= \quad e \mid \neg c \mid c \wedge c \mid c \vee c \mid \text{true} \mid \text{false} \\
\text{stmt} & ::= \quad x := e \\
& \quad | \quad c?
\end{array}$$

Figure 9.2: Syntax of our language

Based on abstract interpretation, our framework is generic: it enables mechanically augmenting any standard dataflow analysis with predicates, regardless of its specific properties. The results stated in this chapter have been formally verified with Coq, an interactive proof management system. We also integrated it into FRAMA-C as a standalone plugin, designed to complement [EVA](#). Our experiments show that predicated analyses over more focused – hence simpler to implement – domains may significantly enhance the precision of the results of [EVA](#), while remaining scalable.

9.2 A GENERIC ABSTRACT INTERPRETATION BASED FRAMEWORK

Predicated analyzers are mostly independent of the target language, even though our implementation handles C programs. For the sake of brevity, we formalize the predicated domains and analyses over a simplified version of `clike`, without pointer values. Thus, the concrete values and the concrete states of its semantics do not involve memory layouts. We still identify the programs and their control-flow graphs, and use a collecting semantics to characterize them. We briefly present the simplified language hereafter.

SYNTAX Figure 9.2 presents the syntax of our language. Programs operate over a fixed, finite set of variables \mathcal{X} whose values belong to an unspecified set $\overline{\mathbb{V}}$. Expressions are either variables, constants, or the application of a binary operator \diamond to expressions. We stratify expressions e in expr and conditions c in \mathbb{C} , the truth value of an element of $\overline{\mathbb{V}}$ being given by a mapping \mathbb{T} from $\overline{\mathbb{V}}$ to booleans. Statements are either assignments such as $x := e$, or tests $c?$ that halt execution when the condition does not hold.

CONCRETE SEMANTICS A concrete state of the program at a node n of its control-flow graph is described by a memory $\mathbf{m} \in \overline{\mathbb{V}}^{\mathcal{X}}$ assigning a value to each variable. The semantics $\llbracket e \rrbracket_{\mathbf{m}}$ (resp. $\llbracket c \rrbracket_{\mathbf{m}}$) of an expression e (resp. a condition c) is its evaluation in the memory \mathbf{m} , and implicitly depends on the semantics of the operators \diamond .

$$\begin{array}{l}
\text{(a) Concrete semantics} \\
\overline{\llbracket x := e \rrbracket}(S) \triangleq \left\{ \mathfrak{m}[x \mapsto \llbracket e \rrbracket_{\mathfrak{m}}] \mid \mathfrak{m} \in S \right\} \\
\overline{\llbracket c ? \rrbracket}(S) \triangleq \left\{ \mathfrak{m} \mid \mathfrak{m} \in S \wedge \mathsf{T}(\llbracket c \rrbracket_{\mathfrak{m}}) = \mathsf{true} \right\} \\
\\
\text{(b) Abstract semantics} \\
\gamma_{\mathcal{L}}(\top_{\mathcal{L}}) = \overline{\mathbb{V}}^{\mathcal{X}} \\
\gamma_{\mathcal{L}}(l_1) \cup \gamma_{\mathcal{L}}(l_2) \subseteq \gamma_{\mathcal{L}}(l_1 \sqcup_{\mathcal{L}} l_2) \\
\overline{\llbracket i \rrbracket}(\gamma_{\mathcal{L}}(l)) \subseteq \gamma_{\mathcal{L}}(\llbracket i \rrbracket_{\mathcal{L}}^{\#}(l))
\end{array}$$

Figure 9.3: Concrete and abstract semantics

The semantics $\overline{\llbracket \text{stmt} \rrbracket}$ of a statement `stmt` is a transfer function over a set of memories, described by the first equalities of Figure 9.3a. After an assignment $x := e$, the variable x is bound (in the new states) to the value of the expression e . A test blocks execution and only allows states in which the condition holds.

ABSTRACT SEMANTICS A predicated analysis relies on a standard abstract domain \mathcal{L} , equipped with the classic operations of the abstract interpretation framework presented in Section 2.2.2:

- a partial order $\sqsubseteq_{\mathcal{L}}$ over abstract states,
- a monotone *concretization* function $\gamma_{\mathcal{L}}$ from \mathcal{L} to $\mathcal{P}(\overline{\mathbb{V}}^{\mathcal{X}})$, linking the abstract states to the concrete ones,
- greatest and smallest elements $\top_{\mathcal{L}}$ and $\perp_{\mathcal{L}}$, such that $\gamma_{\mathcal{L}}(\top_{\mathcal{L}}) = \overline{\mathbb{V}}^{\mathcal{X}}$ and $\gamma_{\mathcal{L}}(\perp_{\mathcal{L}}) = \emptyset$,
- sound over-approximations join $\sqcup_{\mathcal{L}}$ and meet $\sqcap_{\mathcal{L}}$ of the union and intersection of concrete states,
- sound abstract transfer functions $\llbracket \text{stmt} \rrbracket_{\mathcal{L}}^{\#}$ from \mathcal{L} to \mathcal{L} that over-approximate the concrete semantics.

ENTAILMENT AND EQUIVALENCE OF CONDITIONS In the following, we will need to compare some conditions, in particular to decide whether one condition logically implies another. To do so, we choose a coarse interpretation, that treats the expressions present inside conditions as uninterpreted terms. Let Δ be the set $\text{expr}^{\{\mathsf{true}, \mathsf{false}\}}$ of functions from expressions to booleans. Given such a function $\delta \in \Delta$, we lift it to a valuation on conditions in the obvious way, e.g. $\delta(c_1 \wedge c_2) = \delta(c_1) \wedge \delta(c_2)$ where in the r.h.s., the symbol \wedge is the usual conjunction operator. We say that a condition c_1 *entails* another condition c_2 , written $c_1 \vdash c_2$ when the evaluation of c_1 implies the

evaluation of c_2 for all valuations. Similarly, we define the *equivalence* $\dashv\vdash$ of two conditions as their mutual entailment.

$$\begin{aligned} c_1 \vdash c_2 &\triangleq \forall \delta \in \Delta, \delta(c_1) \Rightarrow \delta(c_2) \\ c_1 \dashv\vdash c_2 &\triangleq \forall \delta \in \Delta, \delta(c_1) \Leftrightarrow \delta(c_2) \end{aligned}$$

For example, $((x > y) \wedge (z = 0)) \wedge (h = 2) \vdash (h = 2) \wedge (x > y)$ holds.

As a partial preorder, this entailment remains quite weak. Since it does not give a meaning to the operators \diamond inside expressions, the relation between e.g. $x > 3$ and $x \geq 1$ is not captured, and $x > 3 \vdash x \geq 1$ does *not* hold. This is by design, so that implication and equivalence may be decided efficiently. The real entailment relation may be arbitrarily stronger: any decidable pre-order compatible with $T(\llbracket x \rrbracket_m)$ is also suitable.

9.3 THE PREDICATED DOMAIN

This section shows how to augment a generic abstract domain with conditional predicates. We first define our predicated domain, equip it with a lattice structure, and then define operations suitable for an efficient analysis.

9.3.1 Predicated Elements

Our analysis builds a *predicated* domain on top of any abstract domain \mathcal{L} ; we refer to \mathcal{L} as the *underlying* domain. The information we propagate in this new domain is two-fold:

1. A mapping I from predicates in \mathbb{C} to elements of \mathcal{L} , called a *map*. Maps stand for implications from guards to (abstract) values. Hence they contain *conditional* information: if I maps p to l , then l is a correct approximation of the state as soon as p holds.
2. A boolean predicate $c \in \mathbb{C}$, called the *context*, standing for a set of facts that we know to hold at the current program point. Contexts are used to preserve information when performing a join operation. In particular, the join defined in Section 9.3.3 uses the context to form new interesting guards.¹

We use the syntax $\lambda p.l$ to denote the map from p to l . We write $\langle p \rightarrow l \rangle \in I$ to mean that I guards l by p , and $I(p)$ for l . We say that $\langle p \rightarrow l \rangle$ is *trivial* when $l = \top_{\mathcal{L}}$, as the value $\top_{\mathcal{L}}$ brings no information whatsoever. In order to have a decidable semantics, we restrict ourselves to *finite* maps in which all but a finite number of implications

¹ In our analysis, presented in Section 9.4, contexts are always derived from the guards of the test statements present in the program.

are trivial. This restriction is also important because we often perform seemingly infinite intersections $\bigcap_{p \in \mathbb{C}} I(p)$. In fact, those intersections always involve a finite number of guards p bound to a value different from $\top_{\mathcal{L}}$.

We also require the guard `false`, which corresponds to a contradiction, to be bound to $\perp_{\mathcal{L}}$. In the following, we only mention non-trivial guards, and omit the guard for `false`.²

We call a context and a map that satisfy these properties a *context-implication-map pair*, ranged over by Φ and abbreviated as **CI**-pair. We define $\mathcal{L}^{\text{pred}}$, the *predicated domain over \mathcal{L}* , as the set of such **CI**-pairs. **CI**-pairs will represent the abstract state of our predicated analysis.

The concretization of a **CI**-pair is defined as follows. We say that an implication $\langle p \rightarrow l \rangle$ *holds* in a concrete state m when, if p holds in the concrete state m , then m belongs to the concretization of l . Formally, $\llbracket p \rrbracket_m \Rightarrow m \in \gamma_{\mathcal{L}}(l)$. The concretization $\gamma_{\text{pred}}(c, I)$ of a **CI**-pair is the set of states wherein c is true and all implications of I hold.

$$\gamma_{\text{pred}}(c, I) \triangleq \left\{ m \mid \llbracket c \rrbracket_m = \text{true} \wedge \forall p \in \mathbb{C}, \llbracket p \rrbracket_m \Rightarrow m \in \gamma_{\mathcal{L}}(I(p)) \right\}$$

Notice that the concretization is consistent with our convention for trivial implications, which hold by definition in any concrete state. Therefore, only the non-trivial implications impact the concretization of a **CI**-pair.

9.3.1.1 Rewriting Guards

For the sake of clarity, we use a special notation λ_{\sqcap} to denote the application of a rewriting operator on the *guards* of a map. Given an operator O from guards to guards, applying it naively on an implication map I would lead to “collisions”: distinct guards p_1, \dots, p_n may be rewritten by O into a single guard p . In this case, $O(I)$ should bind p to the meet of all the values previously mapped to p_1, \dots, p_n , i.e. to $I(p_1) \sqcap_{\mathcal{L}} \dots \sqcap_{\mathcal{L}} I(p_n)$. Our notation λ_{\sqcap} makes implicit this meet. Formally, given $f : \mathbb{C}^n \rightarrow \mathbb{C}$ and $l : \mathbb{C}^n \rightarrow \mathcal{L}$:

$$\lambda_{\sqcap}^{\vec{x}}(f(\vec{x})) . l(\vec{x}) \quad \text{means} \quad \lambda p. \bigcap_{\vec{x} \in \mathbb{C}^n} \{l(\vec{x}) \mid p \dashv\vdash f(\vec{x})\}$$

$f(\vec{x})$ should be seen as a *pattern*, that involves the variables bound by \vec{x} , but may also mention other variables bound elsewhere. For instance, to add by conjunction a predicate c to the guard of each implication of a map I , we will write the new map as $I' = \lambda_{\sqcap}^p(p \wedge c) . I(p)$. Here, the notation stands for $\lambda p. \bigcap_{\mathcal{L}} \{I(q) \mid \forall q \in \mathbb{C}, q \dashv\vdash p \wedge c\}$. For any predicate p such that $p \not\vdash c$, the new map binds the predicate $p \wedge c$ to the meet of both previous values $I(p)$ and $I(p \wedge c)$. On the contrary, p is now bound to $\top_{\mathcal{L}}$ (the meet of the empty set) since no predicate q verifies $q \dashv\vdash p \wedge c$.

² By a slight abuse of notation, we also omit the guard for `false` when defining maps through the notation $\lambda p.l$.

9.3.2 Predicated Lattice

Assuming that $(\mathcal{L}, \sqcup_{\mathcal{L}}, \sqcap_{\mathcal{L}})$ is a lattice, we can equip the set of CI-pairs with a derived lattice structure. For convenience, given a CI-pair $\Phi = (c, I)$, we use $\Phi(p)$ for $I(p)$. First and foremost, note that for a given CI-pair Φ and a predicate p , not only does $\Phi(p)$ approximate the concrete states whenever p holds, but so do all the \mathcal{L} -states bound in Φ to weaker guards p' . We can therefore define an even more precise abstraction of the states implied by p by gathering all the abstract states guarded by such a guard p' , and over-approximating their intersection. We call this abstraction *consequence*.

Definition 69. Given $\Phi = (c, I)$, the *consequence* $\Phi \downarrow p$ of p in Φ is defined as:

$$\Phi \downarrow p \triangleq \bigsqcap_{p' \in \mathcal{L}} \{I(p') \mid p \wedge c \vdash p'\}$$

It is immediate that $\Phi \downarrow p \sqsubseteq_{\mathcal{L}} \Phi(p)$ indeed holds for all Φ and p . Also, guards that contradict the context have $\perp_{\mathcal{L}}$ as a consequence, since I maps false to $\perp_{\mathcal{L}}$.

Example 33. In the following examples, \mathcal{L} is a basic interval domain. Consider a CI-pair Φ with the trivial context `true` and two non-trivial implications, $p \rightarrow x \in [2; 6]$ and $q \rightarrow x \in [1; 3]$. Then Φ also carries some information for $p \wedge q$, since $\Phi \downarrow (p \wedge q) = \{x \in [2; 3]\}$. Suppose now that the context of Φ is $p \wedge r$. Then $\Phi \downarrow \text{true} = \{x \in [2; 6]\}$, since $p \wedge r \vdash p$.

Using the consequence operator, we can now define a preorder on CI-pairs, as well as join and meet operations. This will induce a lattice structure on the set CI-pairs. A CI-pair $\Phi_1 = (c_1, I_1)$ is more precise than $\Phi_2 = (c_2, I_2)$, which we write $\Phi_1 \sqsubseteq_{\text{pred}}^{\downarrow} \Phi_2$, when c_1 is stronger than c_2 and all the consequences of Φ_1 are more precise than those of Φ_2 . The join $\Phi_1 \sqcup_{\text{pred}}^{\downarrow} \Phi_2$ has a context equal to the disjunction of c_1 and c_2 , and associates each predicate to the join of its consequences in Φ_1 and Φ_2 . Conversely, the meet $\Phi_1 \sqcap_{\text{pred}}^{\downarrow} \Phi_2$ has a context equal to the conjunction of c_1 and c_2 , and a map obtained by lifting $\sqcap_{\mathcal{L}}$ pointwise. Finally, $\sqsubseteq_{\text{pred}}^{\downarrow}$ establishes naturally an equivalence relation $\sim_{\text{pred}}^{\downarrow}$ on the set of CI-pairs.

Definition 70. Let $\Phi_1 = (c_1, I_1)$ and $\Phi_2 = (c_2, I_2)$.

$$\begin{aligned} \Phi_1 \sqsubseteq_{\text{pred}}^{\downarrow} \Phi_2 &\triangleq c_1 \vdash c_2 \wedge \forall p \in \mathbb{C}, \Phi_1 \downarrow p \sqsubseteq_{\mathcal{L}} \Phi_2 \downarrow p \\ \Phi_1 \sim_{\text{pred}}^{\downarrow} \Phi_2 &\triangleq c_1 \dashv\vdash c_2 \wedge \forall p \in \mathbb{C}, \Phi_1 \downarrow p = \Phi_2 \downarrow p \\ \Phi_1 \sqcup_{\text{pred}}^{\downarrow} \Phi_2 &\triangleq c_1 \vee c_2, \lambda p. (\Phi_1 \downarrow p \sqcup_{\mathcal{L}} \Phi_2 \downarrow p) \\ \Phi_1 \sqcap_{\text{pred}}^{\downarrow} \Phi_2 &\triangleq c_1 \wedge c_2, \lambda p. (\Phi_1(p) \sqcap_{\mathcal{L}} \Phi_2(p)) \end{aligned}$$

We write $\mathcal{L}^{\text{pred}\sim}$ the set of CI-pairs quotiented by the relation $\sim_{\text{pred}}^{\downarrow}$. By construction, two CI-pairs that are equivalent w.r.t this relation

contain exactly the same information. In fact, their concretizations are identical, as stated below:

Lemma 27. *Given two CI-pairs Φ_1 and Φ_2 , $\Phi_1 \sim \Phi_2$ implies $\gamma_{\text{pred}}(\Phi_1) = \gamma_{\text{pred}}(\Phi_2)$.*

Equiped with the operators defined above, $\mathcal{L}^{\text{pred}\sim}$ is itself a lattice, as stated by the following three results.

Lemma 28. *The relation $\sqsubseteq_{\text{pred}}^\downarrow$ is a partial order on $\mathcal{L}^{\text{pred}\sim}$.*

We write \sup (resp. \inf) the least upper bound (resp. greatest lower bound) of two elements of $\mathcal{L}^{\text{pred}\sim}$. Then $\sqcup_{\text{pred}}^\downarrow$ and \sup coincide, $\sqcap_{\text{pred}}^\downarrow$ and \inf coincide, and $\sqsubseteq_{\text{pred}}^\downarrow$ induces a lattice structure over $\mathcal{L}^{\text{pred}\sim}$.

Lemma 29. *$(\mathcal{L}^{\text{pred}\sim}, \sqsubseteq_{\text{pred}}^\downarrow)$ is a lattice, in which*

$$\begin{aligned}\Phi_1 \sqcup_{\text{pred}}^\downarrow \Phi_2 &= \sup(\Phi_1, \Phi_2) \\ \Phi_1 \sqcap_{\text{pred}}^\downarrow \Phi_2 &= \inf(\Phi_1, \Phi_2)\end{aligned}$$

Finally, the predicated join and meet of CI-pairs are respectively over-approximations of the union and intersection of concrete states (with respect to the concretization function).

Lemma 30. *$\sqcup_{\text{pred}}^\downarrow$ and $\sqcap_{\text{pred}}^\downarrow$ are sound:*

$$\begin{aligned}\gamma_{\text{pred}}(\Phi_1) \cup \gamma_{\text{pred}}(\Phi_2) &\subseteq \gamma_{\mathcal{L}}(\Phi_1 \sqcup_{\text{pred}}^\downarrow \Phi_2) \\ \gamma_{\text{pred}}(\Phi_1) \cap \gamma_{\text{pred}}(\Phi_2) &\subseteq \gamma_{\mathcal{L}}(\Phi_1 \sqcap_{\text{pred}}^\downarrow \Phi_2)\end{aligned}$$

We write \top_{pred} and \perp_{pred} for the most general and most restrictive CI-pairs, respectively. Both \top_{pred} and \perp_{pred} contain trivial implications only (except for false).

Definition 71. The greatest and least element of $(\mathcal{L}^{\text{pred}\sim}, \sqsubseteq_{\text{pred}}^\downarrow)$ are respectively

$$\begin{aligned}\top_{\text{pred}} &\triangleq (\text{true}, \lambda p. \top_{\mathcal{L}}) \\ \perp_{\text{pred}} &\triangleq (\text{false}, \lambda p. \top_{\mathcal{L}})\end{aligned}$$

The definition of \perp_{pred} might seem strange, as it would be tempting to bind all predicates to $\perp_{\mathcal{L}}$ instead. However, such a map would not be finite. Furthermore, since the context is false, the contents of the map are actually irrelevant. Indeed, given any map I and predicate p , $(\text{false}, I) \downarrow p = I(\text{false}) = \perp_{\mathcal{L}}$.

9.3.2.1 Computing joins

The definition we have given for $\sqcup_{\text{pred}}^\downarrow$ does not easily lend itself to an implementation. Indeed, our definition uses an universal quantification on all predicates, and the result of a join may contain an

unbounded number of non-trivial implications. (Trivial implications do not contribute to the result of \downarrow , as their bound is $\top_{\mathcal{L}}$.) However, if the two inputs are finite CI-pairs, then there is a finite CI-pair in the equivalence class of the join.

Example 34. Consider two CI-pairs Φ_1 and Φ_2 with the same trivial context `true` and these respective non-trivial implications:

$$\begin{array}{l|l} p \rightarrow x \in [0] & r \rightarrow x \in [42] \\ q \rightarrow x \in [1] & \end{array}$$

Then their join Φ_{\sqcup} has context `true`, and contains at least these implications:

$$p \wedge r \rightarrow x \in [0; 42] \quad q \wedge r \rightarrow x \in [1; 42] \quad p \wedge q \wedge r \rightarrow x \in [42]$$

All weaker or unrelated predicates are bound to $\top_{\mathcal{L}}$, as either Φ_1 or Φ_2 (or both) has no information about them. The three implications above immediately arise from the definitions of $\sqcup_{\text{pred}}^{\downarrow}$. Finally, and this is the key to having a finite join, there exist maps representing Φ_{\sqcup} in which all the other (stronger) implications are trivial ones. Consider, for instance, a predicate s stronger than $p \wedge q \wedge r$. By definition of $\sqcup_{\text{pred}}^{\downarrow}$, it should be bound in Φ_{\sqcup} to $(\Phi_1 \downarrow s) \sqcup_{\mathcal{L}} (\Phi_2 \downarrow s)$, which is equal to $\{x \in [42]\}$. However, since $\Phi_{\sqcup}(p \wedge q \wedge r) \sqsubseteq_{\mathcal{L}} \Phi_{\sqcup} \downarrow s$ by definition, binding s to $\{x \in [42]\}$ is redundant, and it can instead be bound to $\top_{\mathcal{L}}$.

However, notice that we had to consider all combinations of conjunctions of predicates from Φ_1 and Φ_2 to compute Φ_{\sqcup} . In the general case, computing a join is exponential in the number of implications present in its inputs. We let $|\Phi|$ be the number of non-trivial and non-false implications in the CI-pair Φ . There exist CI-pairs Φ_1 and Φ_2 such that $\Phi_1 \sqcup_{\text{pred}}^{\downarrow} \Phi_2$ requires at least $2^{|\Phi_1|+|\Phi_2|}$ implications to be represented.

9.3.3 A Weaker Join

The high complexity of the algebraic lattice structure of CI-pairs would be a serious hindrance to an efficient practical analysis. We construct instead a relaxed join operation. In essence, we define a *weak-join* \sqcup_{pred} which is an upper bound of its arguments, but not the *least* [San+o6a]. Said otherwise, \sqcup_{pred} is an over-approximation of $\sqcup_{\text{pred}}^{\downarrow}$.

Definition 72. Let $\Phi_1 = (c_1, I_1)$ and $\Phi_2 = (c_2, I_2)$ be two **CI**-pairs. The weak-join $\Phi_1 \sqcup_{\text{pred}} \Phi_2$ between them is defined as:

$$(c_1, I_1) \sqcup_{\text{pred}} (c_2, I_2) = (c_1 \vee c_2, \lambda p. (l_{\cup}(p) \sqcap_{\mathcal{L}} l_1(p) \sqcap_{\mathcal{L}} l_2(p)))$$

$$\text{where } \begin{cases} l_{\cup} &= \lambda_{\sqcap}^{(p_1, p_2)} (p_1 \wedge p_2) . I_1(p_1) \sqcup_{\mathcal{L}} I_2(p_2) \\ l_1 &= \lambda_{\sqcap}^{p_1} (\neg c_2 \wedge p_1) . I_1(p_1) \\ l_2 &= \lambda_{\sqcap}^{p_2} (\neg c_1 \wedge p_2) . I_2(p_2) \end{cases}$$

The context of the weak-join remains the disjunction of the prior contexts. Within the implication map, the operator l_{\cup} combines implications of the two previous maps: the \mathcal{L} -join of values present under guards p_1 and p_2 respectively in Φ_1 and Φ_2 is kept under the new guard $p_1 \wedge p_2$. Conversely, the operators l_1 and l_2 preserve the values only present in Φ_1 or Φ_2 respectively. A value l valid in Φ_1 under a guard p_1 may be present in the weak-join under a guard q , provided that the two following conditions hold. First, q must imply p_1 , so that its consequence in Φ_1 is smaller than l . Second, q must contradict c_2 , so that its consequence in Φ_2 is $\perp_{\mathcal{L}}$. Thus, the consequences of q in Φ_1 and Φ_2 are both included in the value l , which can be bound to q in the weak-join. We naturally choose $q = \neg c_2 \wedge p_1$. Symetrically, values present in Φ_2 are present under guards that negate c_1 . Note that this additional information from Φ_i is useless if all guards $p \wedge \neg c_j$ contradict the new context, i.e. whenever $c_i \vdash c_j$.

Example 35. Let us continue Example 34. Operator l_{\cup} creates only the first two implications stemming from the “full” join operator, thus avoiding the potential blow-up of processing all the combinations of conjunctions of predicates from Φ_1 and Φ_2 . Also, the operators l_1 and l_2 do nothing here, as the negation of the contexts of the **CI**-pairs is false. Notice that the loss of precision regarding $p \wedge q \wedge r$ is irrecoverable: knowing $p \wedge r \rightarrow x \in [0; 42]$ and $q \wedge r \rightarrow x \in [1; 42]$ by the weak-join of definition 72, we can only deduce that $p \wedge q \wedge r \rightarrow x \in [1; 42]$. This is strictly less precise than the value $x \in [42]$ obtained with the “strong” join operator.

This is actually a general property of \sqcup_{pred} . The implications “missing” in the weak join always contain a conjunction of several guards from the same map. These are for example the guards of the form $(\bigwedge_i p_i) \wedge (\bigwedge_j p_j)$ with $|i| > 1$ or $|j| > 1$, where the p_i come from one map, and the p_j from the other.

Example 36. Consider now Figure 9.4, that introduces the result of a predicated analysis with the interval domain on the sample code on its left. We write Φ_i for the state at the end of line i , its context and non-trivial implications being shown in the two rightmost columns. We have $\Phi_4 = \Phi_2 \sqcup_{\text{pred}} \Phi_3$ by definition. The value implied by true in Φ_4 comes from the operator l_{\cup} , and is equal to $I_2(\text{true}) \sqcup_{\mathcal{L}} I_3(\text{true})$.

```

1  x = 0; y = 0; v = 1;
   if (c) { x = v; }
   else { y = v; }

5  w = 0;
   if (c)
     { c = 2; }

   ...

```

line	Φ_{line} : state after the statement	
	context	implications
1	true	$\text{true} \mapsto v \in [1], x \in [0], y \in [0]$
2	c	$\text{true} \mapsto v \in [1], x \in [1], y \in [0]$
3	$\neg c$	$\text{true} \mapsto v \in [1], x \in [0], y \in [1]$
4	$c \vee \neg c$ $\equiv \text{true}$	$\text{true} \mapsto v \in [1], x \in [0, 1], y \in [0, 1]$ $c \mapsto v \in [1], x \in [1], y \in [0]$ $\neg c \mapsto v \in [1], x \in [0], y \in [1]$
5	true	$\text{true} \mapsto v \in [1], w \in [0], x \in [0, 1], y \in [0, 1]$ $c \mapsto v \in [1], w \in [0], x \in [1], y \in [0]$ $\neg c \mapsto v \in [1], w \in [0], x \in [0], y \in [1]$
6	c	$\text{true} \mapsto v \in [1], w \in [0], x \in [1], y \in [0]$
7	true	$\text{true} \mapsto v \in [1], w \in [0], x \in [1], y \in [0], c \in [2]$
8	true	$\text{true} \mapsto v \in [1], w \in [0], x \in [0, 1], y \in [0, 1]$ $c \mapsto v \in [1], w \in [0], x \in [1], y \in [0], c \in [2]$

Figure 9.4: Example of an analysis using a predicated interval analysis

Conversely, the value implied by c comes from the operator l_1 , which negates the context of Φ_3 ; furthermore, the value is exactly $\Phi_2(\text{true})$. Note that the intervals inferred in Φ_2 and Φ_3 are entirely retained, guarded by the negations of the converse contexts; no information is actually lost. The figure serves as an example throughout this chapter, and is thus explained in detail afterwards.

The following result states that our weak-join \sqcup_{pred} is weaker than $\sqcup_{\text{pred}}^\downarrow$. Since $\sqcup_{\text{pred}}^\downarrow$ is the least upper bound, \sqcup_{pred} is an upper bound, hence correct.

Lemma 31. : If Φ_1 and Φ_2 are two *CI*-pairs, then

$$\left(\Phi_1 \sqcup_{\text{pred}}^\downarrow \Phi_2 \right) \sqsubseteq_{\text{pred}}^\downarrow (\Phi_1 \sqcup_{\text{pred}} \Phi_2)$$

9.3.3.1 Efficient implementation of the weak-join operation

Consider the definition of $\Phi_1 \sqcup_{\text{pred}} \Phi_2$. The operators l_1 and l_2 are linear on the size of the corresponding map. On the other hand, l_\cup

$$\begin{aligned}
\text{lift}(\mathbf{i}, (c, I)) &\triangleq (c, \lambda p. \{\mathbf{i}\}_{\mathcal{L}}^{\#}(I(p))) \\
\text{kill}(x, (c, I)) &\triangleq (\text{kill}_{\mathcal{C}}^+(x, c), \lambda_{\Pi}^p(\text{kill}_{\mathcal{C}}^-(x, p)) \cdot I(p)) \\
\text{assume}(e, (c, I)) &\triangleq (c \wedge e, \lambda_{\Pi}^p(p[e \leftarrow \text{true}]) \cdot I(p)) \\
\{\!|x := e|\!\}_{\text{pred}}^{\#}(\Phi) &\triangleq \text{lift}(x := e, \text{kill}(x, \Phi)) \\
\{\!|c ?|\!\}_{\text{pred}}^{\#}(\Phi) &\triangleq \text{lift}(c ?, \text{assume}(c, \Phi))
\end{aligned}$$

Figure 9.5: Definition of the abstract semantics $\{\cdot\}_{\text{pred}}^{\#}$

requires $|\Phi_1| \times |\Phi_2|$ operations. Thus, the total complexity is *a priori* in $|\Phi_1| \times |\Phi_2|$.

We can however refine this bound. Any implication $\langle p \rightarrow l \rangle$ present in both Φ_1 and Φ_2 will exist in the join. Thus, any implication of the form $\langle p \wedge p' \rightarrow l \sqcup_{\mathcal{L}} l' \rangle$ is redundant with $\langle p \rightarrow l \rangle$ and does not need to be considered. An optimized implementation of the weak-join should thus consider only the subparts of the maps that are distinct. The practical complexity is now in $|\Phi_1^{\text{diff}}| \times |\Phi_2^{\text{diff}}|$, where $|\Phi_i^{\text{diff}}|$ is the map that contains the implications of Φ_i not present in the other map. This is of particular interest when performing a dataflow analysis: the two maps at a join point share all implications collected before the control-flow split and not modified in-between.

9.4 A PREDICATED ANALYSIS

This section presents the transfer functions used for a dataflow analysis on predicated domains, explains how to avoid the computation of redundant guarded values, and details some strategies to decrease the practical complexity of our analysis.

9.4.1 The Abstract Transfer Functions

As the symbolic equality domain presented in Section 8.2, predicated analyses rely on an operator that computes the dependence of expressions (see Section 8.2.1): $\text{deps}(e)$ is the set of variables on which the evaluation of e depends. On our toy language, this is the set of variables syntactically present in e . However, due to pointer values, $\text{deps}(e)$ usually depends on the current program point in the C language.

Figure 9.5 defines our abstract semantics for statements in $\mathcal{L}^{\text{pred}}$. The gist of the analysis is to apply the transfer functions of \mathcal{L} to each of its elements in the map, which is carried out by the lift function. However, to remain sound, we also need to modify predicates (either in the context or in a guard) whose truth values are possibly modified

$$\begin{aligned}
\text{assume}(\neg e, (c, I)) &\triangleq (c \wedge \neg e, \lambda_{\sqcap}^p (p[e \leftarrow \text{false}]) . I(p)) \\
\text{assume}(p_1 \wedge p_2, \Phi) &\triangleq \text{assume}(p_1, \Phi) \sqcap_{\text{pred}}^{\downarrow} \text{assume}(p_2, \Phi) \\
\text{assume}(p_1 \vee p_2, \Phi) &\triangleq \text{assume}(p_1, \Phi) \sqcup_{\text{pred}}^{\downarrow} \text{assume}(p_2, \Phi)
\end{aligned}$$

Figure 9.6: Extended assume to predicates in disjunctive normal form.

by a statement. Following standard dataflow terminology, we define a kill operator, that removes within predicates the expressions that depend on a certain variable x . This operator is used for an assignment such as $x := e$, as this instruction modifies the value of x . It relies on two $\text{kill}_{\mathbb{C}}$ functions on predicates, whose action depend on whether the predicate occurs in a positive or a negative position. $\text{kill}_{\mathbb{C}}^+(x, p)$ (resp. $\text{kill}_{\mathbb{C}}^-(x, p)$) replaces by `true` (resp. `false`) the sub-expressions of p that depend on x , alternating with the other operator when they encounter the operator \neg .³

While `kill` and `lift` used in conjunction are sufficient to define a sound abstract semantics for $\mathcal{L}^{\text{pred}}$, they never use the existing implications or enrich the context. The join operation retains some of the specific information of each branch (by creating new implications), but only when the branches have different non-true contexts. Thus, we define an operator `assume` that enriches the context by a new expression $e \in \mathbb{C}$, supposed to be satisfied, and replaces by `true` the occurrences of e in the guards of the map. As a side-effect, the value under a guard implied by e gets merged with the value under the guard `true`, refining it. This `assume` operator is extended in Figure 9.6 to predicates in disjunctive normal form: assuming a disjunction amounts to joining the `assume` of each conjunctive clauses, which are themselves the meet of the `assume` of the literals. Assuming the negation of an expression consists in replacing it by `false` in the guards.

Within our abstract semantics $\{\cdot\}_{\text{pred}}^{\#}$, it is natural to use `assume` after a test $c?$, where the predicate c holds by definition. This is exactly what we did in the examples of Section 9.3, to keep track of which branch of a conditional we were in.

Example 37. After line 2 in Figure 9.4, in the branches of the conditional, the operator `assume` enriches the context according to the condition. After the conditional, the context reverts to `true` due to the join between Φ_2 and Φ_3 . At line 6, on a conditional with the same condition c , the `assume` operator maps the `true` guard to $I_5(\text{true}) \sqcap_{\mathcal{L}} I_5(c)$, as c is now `true`. We have re-learned the information known about x and y at line 5. Notice that `assume` removes the guards that are redundant or incompatible with the context, keeping

³ Those operators are formally defined as function `kill_pred` in the Coq proofs.

only the facts relevant at the current program point. On line 7, c is overwritten. Hence, the context c is reset to `true` by the kill operator. Finally, upon exiting the conditional, we lose the information $\neg c \mapsto v \in [1], w \in [0], x \in [0], y \in [1]$ coming from the “else” branch, as negating the context `true` results in an implication that never holds. But the information coming from the “then” branch is preserved under the guard $\neg \neg c$, equivalent to c .

As an optimization not presented in Figure 9.5, it is sometimes useful to *skip* the application of `assume`. Typically, if a preliminary analysis has detected that no part of condition c will never be tested again, there is no point in tracking it. Conversely, since any application of `assume`(p, \cdot) is sound – provided p actually holds – it is sometimes useful to use `assume` after some well-suited assignments.

Consider for example $b := p$ where b is a boolean variable and p a predicate not dependent on b , a common pattern in generated code. We may assume $(b \vee \neg p) \wedge (\neg b \vee p)$ after such a statement. Then, on a test $b?$, the analysis will be able to re-learn p . Using the `assume` function more or less aggressively can be seen as a trade-off between precision and complexity—in particular because contexts are used by our weak-join operation to create new implications.

9.4.1.1 Soundness of the Analysis

The analysis we have defined above correctly approximates the concrete semantics of the program.

Lemma 32. *Our predicated analysis over $\mathcal{L}^{\text{pred}}$ is sound.*

$$\begin{aligned} \gamma_{\text{pred}}(\Phi_1) \cup s\gamma_{\text{pred}}(\Phi_2) &\subseteq \gamma_{\text{pred}}(\Phi_1 \sqcup_{\text{pred}} \Phi_2) \\ \overline{\{\mathbf{i}\}}(\gamma_{\text{pred}}(\Phi)) &\subseteq \gamma_{\text{pred}}(\{\mathbf{i}\}_{\text{pred}}^{\#}(\Phi)) \end{aligned}$$

Moreover, we can state a stronger result, that links, at a program point n , the abstract semantics of \mathcal{L} with its counterpart on $\mathcal{L}_{\text{pred}}$.

Lemma 33. *If the underlying transfer functions are monotonic, our predicated analysis is as precise as the non-predicated one.*

Of course, the predicated analysis can be more precise. As an example, on line 6 of the program of Figure 9.4, the non-predicated analysis would have inferred the value $I_5(\text{true})$. Our own result—namely $I_6(\text{true})$ —is much more precise.

9.4.2 Improving the Analysis: Avoiding Redundant Values

Amongst the values guarded in the implications of a map I , the value under the `true` guard plays a special role. $I(\text{true})$ always holds by definition, and represents the broadest, less precise knowledge we

have on the state. All other values can be used to refine this value, under some hypothesis. Indeed, whenever a predicate p is satisfied, the meet between $I(\text{true})$ and $I(p)$ is a correct abstraction of the state, more precise than $I(\text{true})$.

Based on this reasoning, all information carried by $I(\text{true})$ can be removed from the other values without any loss of precision. Furthermore, the guarded values may be seen as complementing $I(\text{true})$, and can be handled differently. In particular, the transfer functions of the underlying domain may be expensive – even more so if they are precise. Applying them under each guard is likely to be costly, and may uselessly duplicate some information in each implication of the map.

Reducing the size of the guarded values, as well as the cost of treating them, is essential to decrease the practical complexity of the predicated analysis. For this purpose, we require two additional features from the underlying domain \mathcal{L} .

1. A transfer function $\llbracket i, p \rrbracket_{\mathcal{L} \times \mathbb{C}}^\#$ over statements i , parameterized by the predicate p that guards the processed value. This way, the analysis can be more precise on the true guard only and avoid the duplication of new information. Thus, $\llbracket i, \text{true} \rrbracket_{\mathcal{L} \times \mathbb{C}}^\#$ may be defined as $\llbracket i \rrbracket_{\mathcal{L}}^\#$, while $\llbracket i, \cdot \rrbracket_{\mathcal{L} \times \mathbb{C}}^\#$ applied to any guard other than true should be defined as a very imprecise operation, that only guarantees the soundness of the analysis on \mathcal{L} . Formally, we only require $\llbracket i, \cdot \rrbracket_{\mathcal{L} \times \mathbb{C}}^\#$ to be an over-approximation of $\llbracket i \rrbracket_{\mathcal{L}}^\#$. The lift operator is then redefined as

$$\text{lift}(i, (c, I)) \triangleq (c, \lambda p. \llbracket i, p \rrbracket_{\mathcal{L} \times \mathbb{C}}^\#(I(p)))$$

2. A difference operation $\setminus_{\mathcal{L}}$ that discards information already contained in another element of \mathcal{L} , that we use to simplify implication maps. Ideally, $a \setminus_{\mathcal{L}} b$ should be as large as possible (w.r.t. $\sqsubseteq_{\mathcal{L}}$), while retaining all the information of a not already present in b . To be sound, we require $a \sqsubseteq_{\mathcal{L}} a \setminus_{\mathcal{L}} b$. We define an operator *reduce*, that simplifies each implication w.r.t. the value mapped in the true guard. It can be used at any time, but it is most useful whenever the shape of the map has changed significantly and redundancies may have been introduced (i.e. after a join or an assumption).

$$\text{reduce}(I) \triangleq \lambda p. I(p) \setminus_{\mathcal{L}} I(\text{true})$$

These two operators may discard a lot of information; ideally, they would just keep the values that the non-predicated analysis fails to compute. In fact, provided that $\setminus_{\mathcal{L}}$ is such that no information is irrecoverably lost by its application, then *reduce* actually preserves the information contained in the map: only its actual *contents* are altered.

1	$x = 0; y = 0; v = 1;$
	$\text{if } (c) \{ x = v; \}$
	$\text{else } \{ y = v; \}$
5	$w = 0;$
	\dots

line	context	implications after the statement
4	$c \vee \neg c$ $\equiv \text{true}$	$\text{true} \mapsto v \in [1]; x \in [0, 1]; y \in [0, 1]$ $c \mapsto x \in [1]; y \in [0]$ $\neg c \mapsto x \in [0]; y \in [1]$
5	true	$\text{true} \mapsto v \in [1]; w \in [0]; x \in [0, 1]; y \in [0, 1]$ $c \mapsto x \in [1]; y \in [0]$ $\neg c \mapsto x \in [0]; y \in [1]$

Figure 9.7: Analysis of Figure 9.4 with factorization.

Lemma 34. Suppose that $(a \setminus_{\mathcal{L}} b) \sqcap_{\mathcal{L}} b = a$ holds for all $a, b \in \mathcal{L}$. Then we have $(c, I) \sim_{\text{pred}}^{\downarrow} (c, \text{reduce}(I))$.

Example 38. Let us come back to the example of Figure 9.4. The join and the lift function duplicate the value of variables v and w at line 4 and 5 respectively. Here, our previous analysis kept more information within implications than needed. The benefit of the improvements described above are shown in Figure 9.7. When joining the values coming from lines 2 and 3, the reduce operator removes under the guards c and $\neg c$ the information about v , which is already present under the weaker guard true . In parallel, at line 5, the modified lift operator does not apply the full interval analysis to the values guarded by c and $\neg c$. Instead, we use a simpler abstraction, that only removes information about variables that are overwritten. This way, the information about w is no longer duplicated.

Finally, the new transfer functions $\{\mathbf{i}, \cdot\}_{\mathcal{L} \times \mathbb{C}}^{\#}$ should always have access to the special value $I(\text{true})$. Thus, the transfer functions may rely on $I(\text{true})$ for the parts of the processed value that have been removed by the difference operator.

9.4.2.1 Application to standard domains

A non-relational domain, such as the interval domain, is usually an environment that maps variables (or more complex memory locations) to abstract numeric values. For such a domain, the difference operation can be implemented pointwise, by dropping the bindings already included in the reference state, namely $I(\text{true})$.

For all predicates p , a sound transfer function $\{\mathbf{i}, p\}_{\mathcal{L} \times \mathbb{C}}^{\#}$ has to remove the numeric values bound to the variables that are possibly modified by the statement \mathbf{i} . But they should avoid creating new

bindings in a state guarded by a predicate $p \not\equiv \text{true}$. A worthwhile trade-off could be to add more information to a guarded state only when the statement involves variables present in this particular state. Indeed, thanks to the application of *reduce*, if such a state contains a mapping for a given variable, then its numeric value is more precise than the one bound in the state $I(\text{true})$. Thus, the evaluation of the statement may really be more precise in the guarded state. Note that this last refinement is not implemented in the domains used for our experimental evaluation.

These two optimizations have another advantage. Indeed, they decrease the size of the guarded values and postpone their alteration until it becomes necessary. Thus, the implications collected before a split of the control-flow graph are more likely not to be modified in the branches. This maximises the shared subparts of the **CI**-pairs propagated through parallel branches. Since our efficient join only considers the distinct subparts of maps to create new implications, maximizing the shared subparts prevents an unnecessary increase in the size of the predicated maps.

For relational domains, the principles of those improvements would be the same, but the difference operation may be more difficult to implement. However, it does not have to be complete: it is always sound to retain some redundancy.

9.4.3 Propagating Unreachable States

Whenever the lifted transfer function of the underlying domain returns the abstract state $\perp_{\mathcal{L}}$ for a value kept under a guard p , there is by definition no concrete state where p holds. We can thus refine the **CI**-pair by assuming the predicate $\neg p$. The former and latter maps are not equivalent, as their context differ. However, their concretization is exactly the same, as stated by the result below.

Lemma 35. *Given Φ and p such that either $\Phi(p) = \perp_{\mathcal{L}}$ or $\Phi \downarrow p = \perp_{\mathcal{L}}$, then $\gamma_{\text{pred}}(\Phi) = \gamma_{\text{pred}}(\text{assume}(\neg p, \Phi))$*

Through this mechanism, information can flow from the underlying domain to the predicated one, by means of the contrapositive of the collected implications.

This also means we could embed the context of our abstract states directly in the implication map. A domain mathematically isomorphic to ours is obtained simply by mapping the negation of the context to bottom. However, we chose a formalisation that keeps separate the context and the implication map. For the sake of clarity firstly, as these two components play very different roles in the analysis. Secondly, this design provides more leeway in the implementation, in particular to select finely the predicates of the context.

9.4.4 Convergence of the Analysis

Throughout the analysis of a given program, all guards of non trivial implications present in a map are derived from the conditionals of the program, so their number remains finite. In practice, this number can be high; we discuss a possible way of limiting it in Section 9.7. The predicated analysis essentially amounts to performing the underlying analysis over the values under each guard (except for the assume operations, which allow us to be more precise). Thus, if the underlying domain provides (or requires) a widening operator to effectively compute the fixpoint, then it can (and should) be lifted as well. Finally, if the underlying transfer functions are monotonic, so are the predicated ones, which ensures the termination of our analysis.

9.5 A VERIFIED SOUNDNESS PROOF

The lattice structure of the predicated domain, the relation between the join and the weak-join and the soundness of the analysis have been formalized and proven in Coq [Coq], an interactive theorem prover. The proofs scripts are mechanically checked by the Coq kernel, ensuring their correctness. This increases very significantly the confidence in our formalization. In particular, the soundness of the weak-join operator, and of its optimized version, was a non-trivial result.

This section gives a brief outline of this development. A correspondence between the notations of this manuscript and the Coq ones is available at the beginning of the script.

9.5.1 Prerequisites

Our Coq development is parameterized by the following elements, that are kept abstract.

EXPRESSIONS AND ENVIRONMENTS. We require three sets, standing for numeric values `Value`, variables `Var`, and expressions `Exp`. Concrete states are environments in $\text{Env} := \text{Var} \rightarrow \text{Value}$. Given an environment, the evaluation function `eval_expr` assesses the value of an expression. The update function models the assignment of a single variable in an environment, and the `deps` function verifies that if an expression `exp` does not depend on a variable `var`, then no update of `var` can affect the evaluation of `exp`.

LATTICE. We require a lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ – the underlying abstract domain – plus the correctness of its operations. The Coq development requires a *complete* lattice. Indeed, the join and meet are infinitary, and have type $(L \rightarrow \text{Prop}) \rightarrow L$; the first argument

denotes the set of elements of L that are being joined or met.⁴ This was done to simplify the proofs, as using a binary join or meet would have required to reason on the order in which the operations are performed.

ANALYSIS OVER \mathcal{L} . We require a monotonic concretization function `concr` from L to Env , and monotonic transfer functions `assign` and `assume` with the properties of definition 8.

9.5.2 Lattice Structure

We define the predicates as an inductive structure over the expressions, and extend the evaluation on expressions to predicates. The entailment \vdash and equivalence $\dashv\vdash$ between predicates are defined using this evaluation. We then show that the constructors `LAnd`, `LOr` and `LNot` for predicates are *morphisms* of the relations induced by \vdash and $\dashv\vdash$, and register those lemmas in the type classes mechanism of Coq [SO08]. Then, given $p \dashv\vdash q$, we can prove $(p \wedge r) \dashv\vdash (q \wedge r)$ directly by a rewriting step. This mechanism is used extensively throughout the proofs.

CI-pairs are records of a predicate (the context) and a function from predicates to L (the map). To simplify some definitions, we do not require `false` to be always bound to $\perp_{\mathcal{L}}$. Instead, we prove that all abstract operations preserve this property, called `false_bottom` in the development.

The function `in_map ci P` gathers the values of L bound to guards of `ci` that satisfy the proposition (on predicates) P . The definition of the consequence $\text{ci} \downarrow p$ is then:

$$\sqcap (\text{in_map ci } (\text{fun } p' \Rightarrow p \wedge (\text{context ci}) \vdash p'))$$

The inclusion `CI_incl`, equivalence `CI_equiv`, join `CI_join` and meet `CI_meet` of **CI**-pairs are then defined as specified in Section 9.3, together with the proofs of their correctness: `CI_incl` is an order relation, `CI_equiv` an equivalence relation, `CI_join` the least upper bound and `CI_meet` the greatest lower bound of two **CI**-pairs. These proofs heavily rely on intermediate lemmas about consequences.

9.5.3 Weak-Join

Proving the correctness of the weak-join —and of its optimization— is the more involved part of the development.

The weak-join of **CI**-pairs is defined as the meet of three **CI**-pairs, which correspond to the three operators l_{\cup} , l_1 and l_2 of Definition 72. The first one is created by the function `CI_conj_join`, and the latter two are symmetrically created by the same function `CI_neg_join`.

⁴ Readers unfamiliar with Coq can simply see `Prop` as the set of booleans.

Given Φ_1 and Φ_2 , their weak-join is the meet of $\text{CI_conj_join } \Phi_1 \ \Phi_2$, $\text{CI_neg_join } \Phi_1 \ \Phi_2$ and $\text{CI_neg_join } \Phi_2 \ \Phi_1$. We prove that the results of CI_conj_join and CI_neg_join are greater (less precise) than the original join. Therefore, so is their meet, and the weak-join is indeed greater than the join.

We then validate the final optimization of the weak-join. $\text{CI_shared } \text{ci1 } \text{ci2}$ contains implications belonging to both ci1 and ci2 ; other predicates are bound to \top . Conversely, $\text{CI_diff } \text{ci1 } \text{ci2}$ contains only the elements of ci1 mapped to different values in ci2 ; other predicates are bound to \top . The efficient weak-join is the meet between the shared [CI-pair](#) and the previous weak-join of the rests. The last lemma asserts the equality between the former and the efficient weak-join.

9.5.4 Analysis

The concretization CI_concr links [CI-pairs](#) to concrete environments. We prove that this concretization is monotonic, and consistent with the join and meet operations. The deps function is also extended to predicates. We then define a function kill_pred that implements the two operators kill_C^+ and kill_C^- , defined respectively as weaken_pred and strengthen_pred .

Then we define the transfer function CI_assign and CI_assume such as specified in Section [9.4.1](#). Finally, we ensure the soundness of their definition:

- If the environment env is a concretization of ci , and if the expression exp evaluates to val in env , then $\text{update var val env}$ is a concretization of $\text{CI_assign var exp ci}$;
- If the environment env is a concretization of ci , and if the expression exp evaluates to a positive value in env , then env is a concretization of CI_assume exp ci .

The soundness of CI_assign does not depend on the exact definitions of the functions weaken_pred and strengthen_pred on predicates. We actually prove that more involved operators kill_C are also sound. For example, we could use two operators that invert assignments of the form $x := x + k$ in the guards, instead of removing all the occurrences of x .

Those generalized operators are called weaken and strengthen in the formalization, and take as additional argument the expression to which the variable is being assigned. Let us recall that the operator

$T(\cdot)$ injects values of $\bar{\mathbb{V}}$ into booleans. The properties that must be satisfied by `weaken` and `strengthen` are as follows:

$$\forall c \in \mathbb{C}, m \in \mathbb{M}, x \in \mathcal{X}, e \in \mathbf{exp},$$

$$\begin{cases} T(\llbracket c \rrbracket_m) \Rightarrow T(\llbracket \text{weaken}(c, x, e) \rrbracket_{m[x \mapsto \llbracket e \rrbracket_m]}) \\ T(\llbracket \text{strengthen}(c, x, e) \rrbracket_{m[x \mapsto \llbracket e \rrbracket_m]}) \Rightarrow T(\llbracket c \rrbracket_m) \end{cases}$$

Given the assignment $x := x + 3$, taking for `weaken` the function that replaces occurrences of x by $x - 3$ is obviously correct here. The drawback of this approach is that a potentially infinite number of new predicates may be created, which might lead to a non-terminating analysis. Performing widening steps on the context and the guards might be required.

9.6 RELATED WORK

Convex numeric domains, such as intervals, polyhedra, octagons and linear equalities, are widely used in abstract interpretation. Their convexity enables scalable analysis but impedes the representation of disjunctive invariants, causing overly wide imprecisions. Therefore, a large body of work has been devoted to remedy this shortcoming. *Disjunctive completion* [CC79b; GR98] of abstract domains avoids the computation of joins by propagating multiple abstract states in parallel along the analysis. One downside is that the code may need to be fully analyzed for each separate state, whereas our framework strives to minimize the unnecessary computations by getting rid of redundancy. On the other side, disjunctive completion can be used to unroll loop symbolically, something our approach does not handle. However, as widening is notably hard to perform properly on disjunctive sets [BHZo4], most of these analyses operate on a beforehand bounded number of disjunct states. Then, some join must eventually be performed, and a distance between abstract states [San+o6b; PCo6] can be used to first rejoin the most related states. This is linked to our difference operator, since nearby states (according to this distance metrics) should have small differences.

Additional information can be attached to the disjunct components. In practice, such disjunctive domains are often stored by binary decision-diagram (BDD) [Bry86] where the nodes contain some predicates and the leaves are numeric abstract states.

Boolean partitioning [Ber+10] distinguishes the numerical values of small sets of variables with respect to the truth values of some boolean variables. Such a partitioning may include several boolean trees working on different sets of variables, chosen by heuristics. By comparison, our predicated domain is built on entire states of the underlying domain. Indeed, we do not restrict the variables that may appear in our abstract states. This could be an interesting extension to further

improve scalability. Under a guard p , we could choose to keep information only on the variables that are read or written inside an if whose condition involves p . The *Binary Decision Tree Abstract Domain*, proposed by Chen and Cousot [CC15], uses the conditionals of the program as nodes for the tree, as in our analysis. In their work, the shape of the tree is mostly static, making the join operation simpler to implement. The transfer function for assignments preserves all nodes, unlike in our approach where some guards are weakened or removed. On the other hand, the whole tree must be rebuilt after assignments, which may be very costly.

Trace partitioning [HT98; MR05] associates each component with some set of execution paths, and involves heuristics on the control-flow to choose a partition of the traces that guides the disjunction. Also, traces should be merged when it is no longer useful to keep them separate; syntactic criteria are used to detect such merge points. *Property simulation* [DLS02] avoids the cost of full path-sensitivity for proving a single fixed property: it groups the abstractions of execution states wherein the given property has the same state. The *boxes domain* [GC10a] implements a specific disjunctive refinement of intervals with decision diagrams extended over linear arithmetic, while our framework is parametrized by the underlying domain under consideration. Closer to our approach, although focusing on termination proof through backward analysis, [UM14] designs a decision tree abstract domain from linear constraints to generic values. Some effort is also made to maintain a canonical representation of the trees. However, unlike our setup, the ordering and the join are point-to-point operations relying on unification of trees. Also, widening must be used, as the height of the trees height is not bounded a priori.

While binary decision trees make choices on the truth values of boolean predicates, the *Segmented Decision Tree Domain* [CCM10] can express properties depending on the range of values of arithmetic variables. Each node is a disjunction over exclusive value intervals for a variable, specified by a symbolic segmentation, and the number of possible choices is not bounded *a priori*. Since the number of segments may grow indefinitely, the widening operator must act on the shape of the tree.

As disjunctive completions, all these domains make a strict partition of their abstract states. Each component of the disjunction is the only abstraction of the concrete states for some cases, and the join between them are postponed as much as possible. Therefore, the analysis can not be relaxed on some disjunct without losing precision for the cases it represents. In contrast, our framework performs the join immediately, but preserves the lost information in abstract values guarded in implications. Then, these separate values provide some additional information to the join, but are not intended to be interpreted alone. This design allows the optimizations proposed in Sec-

tion 9.4.2, where the treatment of these ancillary values is lightened. This would be impossible to implement on a disjunctive domain without leading to arbitrary precision loss.

Predicate abstraction [GS96] would infer a single fact along all execution paths, but this fact may be arbitrarily complex and thus can express disjunctive properties. CEGAR [Cla+00] improves predicate abstraction by refining the invariants inferred using counter-examples. Different approaches have been proposed to combine numeric domains with predicates [GC10b; BHT08; FJM05]. In particular, Fischer et al. show how to build analyses that propagate a map from a determinate set of predicates to the numerical elements of any existing dataflow analysis [FJM05]. As usual in counter-examples based techniques, the predicates are incrementally found by successive refinement iterations of the analysis, that prune out unverified invariants. Still, finding the proper predicate may be arbitrarily complex, resulting in hard to predict analysis times. Also, the refinement phase requires decidable theories and powerful decision procedures to find the counter-examples from which the predicate is deduced. We instead chose to limit ourselves to uninterpreted predicates relating the conditionals present in the program, for simplicity and predictability. Furthermore, predicate abstraction is mostly goal-driven, and used by model-checkers to prove that a certain property is valid. Our predicated framework uses predicates to postpone the loss of precision inherent to joins in abstract interpretation, but it is not goal-driven. Instead, the same analysis will be done for e.g. all the potential runtime errors of the program (which is why the analysis needs to be run only once). In particular, improving an unsufficiently precise analysis requires designing more fine-grained analysis domains. This contrasts with the automatic refinement available with predicate abstraction.

Otherwise, Mihaila and Simon present in [MS14] another way to synthesize predicates by observing losses in abstract domain. They propagate a single numeric state augmented with sets of implications between predicates, specifically generated by the numerical abstract domain at join points. For the domain of intervals, the join between the two states where $x \in [0; 5]$ and $x \in [10; 15]$ would typically produce the implication $x > 5 \Rightarrow x \geq 10$. At conditionals in the control-flow graph, implications are fed to the underlying domain to recover the numeric loss. The transfer functions follow the same general considerations than ours, but the predicates stem from the numeric domain and are not restricted to the conditionals of the program. Thus they also heed to avoid generating redundant implications, although this makes the full recovery more intricate than in our construction.

Finally, combining abstract domains is a standard way to enhance the abilities of static analysis based on abstract interpretation. These analyses were introduced in the founding papers [CC79b] and widely studied since [CCF13]. In particular, our predicated domain is a *re-*

duced product between contexts and maps, and the maps can be seen as instances of a *reduced cardinal power* [CC79b], where the base is the chosen underlying abstract domain and the exponent is the set of conditional predicates.

9.7 EXPERIMENTAL RESULTS

We have integrated our predicated analyses framework as a new plugin of the FRAMA-C platform. This plugin complements the results of the Value Analysis (previously [VALUE](#), and now [EVA](#)). We used it on two simple domains presented in the previous section; the obtained results are presented in this section.

9.7.1 *Scope of the Current Implementation*

To limit the imprecisions caused by the junctions of convex abstractions, [EVA](#) already embeds an instance of trace partitioning: it postpones the joins of abstract states by propagating multiple ones separately. As dissociating every feasible execution path leads to intractable analyses, the maximum number of parallel states maintained by [EVA](#) is limited by a parameter called *slevel*. Once this threshold is reached, all new states are joined and propagated without trace partitioning. Still, high *slevel* values may lead to high analysis time.

For performance reasons, this partitioning is directly integrated on the dataflow analysis. It is systematically applied on top of all the domains on which is instantiated the analysis. Unlike the value and state abstractions, this trace abstraction is not extensible for the time being. Instead, we have integrated our predicated analyses framework as a new plugin of the FRAMA-C platform. This plugin complements the [EVA](#) analysis by validating a posteriori some of the alarms it has emitted. By construction, our plugin mainly improves [EVA](#)'s results on successive test statements with identical conditions⁵. Although such pattern is relatively unusual in idiomatic C code, it is much more frequent in generated programs, for which our method is well adapted.

This predicated analyses plugin has been designed to be modular. It is parameterized by the underlying abstract domain, and builds a dataflow analysis with predicates for this domain. This plugin runs after [EVA](#), which it mainly uses to get aliasing information on pointers. This information is needed to ensure the soundness of the deps operator. For convenience, we also chose to reuse some data structures of [EVA](#). In particular, the maps from predicates to abstract values are patricia trees with hash-consing. Finally, all the predicates are

⁵ Modulo conjunction, disjunction and negation, but only over uninterpreted expressions.

normalized into a disjunctive normal form. This way, a **CI**-pair never manages different equivalent predicates.

Generated programs can include a very large number of nested conditional branches and loops, leading to overly wide contexts in our own analysis. To avoid a complexity explosion, we limit the number of literals in the predicates used in contexts and guards (thereby decreasing the precision of our results), according to a parameter level. The join removes any implications whose guard exceeds this limit; in the context, we try to keep the most recent predicates.

The analysis is modular: it processes once and separately each function of a program, retains a summary of their effect regardless of their call context and still benefit from it at each function call. The kill operator, applied to the set of variables potentially modified by the call, guarantees the soundness of our analysis, and the meet of the predicated domain put together the abstract states before the call and at the return statement of the called function.

The plugin is available at <http://yakobowski.org/predicated.html>.

9.7.2 Application on two Simple Domains

This subsection describes the two abstract domains on which we have instantiated our predicated analysis plugin. Of course, our framework could also be applied to other domains, e.g. intervals or the “valid file descriptors” domain used for Figure 9.1.

9.7.2.1 Tracking Initialized Variables

Our experiments relied on a domain retaining, at each program point, the set of variables that were properly initialized. This domain can be used to prove that no uninitialized variables are read at execution time. We used it successfully on generated C programs. In this kind of code, variable initialization may happen inside conditionals, and far away from the points where the variable is used.

In the semantics of our simplified language, we introduce a default value `uninit` in \overline{V} , to which all variables are equal at program entry. The execution of a statement is correct when all the involved variables are initialized.

$$\begin{aligned}\gamma_{\text{init}}(V) &= \{m \mid \forall x \in V, m(x) \neq \text{uninit}\} \\ \{x := e\}_{\text{init}}^{\#}(V) &= \begin{cases} V \cup \{x\} & \text{if } \text{deps}(e) \subseteq V \\ V \setminus \{x\} & \text{otherwise} \end{cases} \\ \{c?\}_{\text{init}}^{\#}(V) &= V\end{aligned}$$

$$\begin{aligned}
\llbracket c?, p \rrbracket_{\text{eq} \times \mathbb{C}}^{\#}(E) &\triangleq \begin{cases} E \cup \{e_1 = e_2\} & \text{if } p \equiv \text{true and } c = (e_1 = e_2) \\ E & \text{otherwise} \end{cases} \\
\llbracket x := e, p \rrbracket_{\text{eq} \times \mathbb{C}}^{\#}(E) &\triangleq \begin{cases} \text{kill}_{\text{eq}}(x, E) \cup \{x = e\} & \text{if } p \equiv \text{true and } x \notin \text{deps}(e) \\ \text{kill}_{\text{eq}}(x, E) & \text{otherwise} \end{cases} \\
\text{kill}_{\text{eq}}(v, E) &\triangleq \{(a = b) \in E \mid v \notin \text{deps}(a) \wedge v \notin \text{deps}(b)\} \\
E \setminus_{\text{eq}} F &\triangleq \{(a = b) \in E \mid (a = b) \notin F\} \\
\gamma_{\text{eq}}(E) &\triangleq \{\mathfrak{m} \mid (a = b) \in E \Rightarrow \llbracket a \rrbracket_{\mathfrak{m}} = \llbracket b \rrbracket_{\mathfrak{m}}\}
\end{aligned}$$

Figure 9.8: Abstract semantics for the equality domain

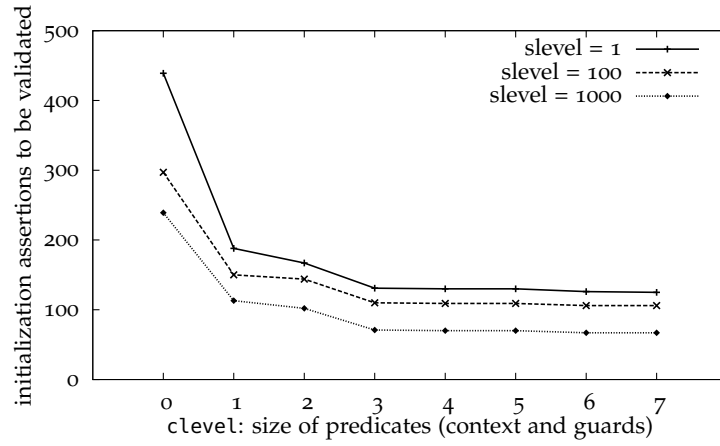
9.7.2.2 A Second Abstract Domain: Herbrand Equalities

We also reused for our experiments the symbolic domain tracking Herbrand equalities between C expressions, presented at Section 8.2. This equality domain boils down to retaining equalities stemming from assignments or equality conditions. Its formal definition is recalled in Figure 9.8, where the set E of equalities increases on tests involving an equality, and on assignments that do not refer to the variable being modified. To be sound, the transfer function on assignments must also remove equalities that involve the overwritten variable, through the kill_{eq} operator. Following Section 9.4.2, we present simplified transfer functions, for which only the true guard is enriched, and in which the operator \setminus_{eq} can be used to remove redundant equalities.

This domain lends itself to a natural extension of our analysis, namely the strengthening of the context and the guards by backward-propagating information from \mathcal{L} . For example, we can quotient all the predicates of the CI-pairs by the congruence relation induced by the equalities stored in the map. Furthermore, when applying the operator kill (following an assignment, say to x) on an expression e that involves x , we may instead substitute e by another equal expression. This is more precise than removing the occurrences of x in the guards and the context.

9.7.3 Results on Variables Initialization

We tested our plugin on a C program of 5800 lines generated by the industrial environment *SCADE*, devoted to real-time software. As often with such codes, multiple conditionals are heavily used —typically to test automata states or clocks. In fact, this program has an extremely high ratio of conditionals w.r.t. the total number of statements: 819 out of 2830. Furthermore, many conditionals are complex, with multiple conjunctions and disjunctions. If the operators $\&\&$ and



slevel	assertions to be validated	initialization assertions	remaining assertions/clevel					
			1	2	3	4	5	6
1	632	439	188	167	131	130	130	126
100	488	297	150	144	110	109	109	106
1000	430	239	113	102	71	70	70	67
2000	409	218	100	96	65	64	64	61

time of the Value Analysis

slevel	time
1	1.4s
100	6.7s
1000	175s
2000	400s

time of the predicated analysis

clevel	time
1	0.7s
2	0.73s
3	0.8s
4	1.0s
5	1.9s
6	10s
7	85s

Figure 9.9: Experimental results

|| are desugared into multiple `if`, the resulting program has 9576 statements, and 3428 conditionals. Thus, this program is a very good benchmark for an analysis.

Our results are presented in Figure 9.9. We first applied [EVA](#), which emitted various assertions it could not validate (column 2). As expected, a higher `slevel` results in fewer alarms. Between 55% and 70% of those are assertions requiring variables to be properly initialized (column 3), which are those the domain of Section 9.7.2.1 understands. We then ran our predicated analysis, instantiated by this domain, with different limits for the size of predicates. For the values of `clevel` we used, the number of initialization assertions still unproven after the predicated analysis are shown in the five corresponding columns. Hence, lower curves are better. The analysis time for the [EVA](#) and for the predicated analysis are also given, according to their parameter; they are independent from each other. Remember that [EVA](#) must be run (at least with `slevel` 1) before our analysis can run. So the two timings should be added to compare the total analysis time.

While [EVA](#) produces significantly less alarms with a higher `slevel`, its analysis time also increases drastically. This is unsurprising, as fully partitioning for k successive conditionals may require as much as 2^k distinct states. On the other hand, our plugin is effective to quickly validate numerous assertions left unproven by [EVA](#), even with strongly limited predicates. The precision of our analysis increases rapidly with the `clevel` parameter, while the analysis time remains reasonable. More generally, it turns out that small contexts are sufficient to retain most of the relevant information: fewer assertions remain to be validated with `clevel` = 1 (2.1s) and `slevel` = 1 than with `clevel` = 0 and `slevel` = 2000 (400s). Intuitively, even inside deeply nested conditionals (which generate complex contexts), the most recent guards seem to be the most useful. The results of our experiments show that it is much more cost efficient to increase the `clevel` parameter than the `slevel` parameter. Those results are extremely encouraging, given the difference in maturity between our plugin and [EVA](#). Indeed, the main abstract domain of [EVA](#) has been considerably optimized for speed for many years.

9.7.4 Validation of the Optimizations

The relevance of the improvements developed in Section 9.4.2 have been validated through some experiments on the same code as above. We compared the efficiency of the optimized predicated analysis (*Opt* in the results) with two modified versions of our framework:

- one without any difference operation, where each relevant value is kept unreduced in the join (*Diffless*);

<i>clevel</i>	Initialization			Equalities			
	Opt	Diffless	OrigTF	Opt	Diffless	OrigTF	
1	0.7s	0.7s	0.9s	2.2s	11s	28s	time
	10	10	10	187	1751	1786	average
	22	22	22	304	3895	4320	max
2	0.73s	0.74s	2.9s	3.7s	58s		time
	25	26	97	391	6983	timeout	average
	44	45	302	680	18653		max
3	0.8s	1.0s	30s	8.4s	240s		time
	89	101	777	1015	23616	timeout	average
	157	172	2732	2269	73518		max
4	1.1s	1.6s		26s			time
	279	382	timeout	3195	timeout	timeout	average
	552	601		9258			max
5	1.9s	4.8s		160s			time
	847	1255	timeout	9004	timeout	timeout	average
	1132	1764		46386			max

- Opt : Optimized analysis as described in the manuscript
 Diffless : Analysis without the difference operation
 OrigTF : Analysis with the original transfer functions applied
 in each implication

Figure 9.10: Effectiveness of the optimizations

- one in which the original transfer function of the underlying abstract domain is applied to each abstract value in the implications (*OrigTF*).

We used as underlying domains the two domains presented in Section 9.7.2. For each configuration and for different size limits for predicates, the Figure 9.10 shows the analysis time and some measure of the amount of information propagated during the analysis. With the domain of initialized variables, we give the average and the maximum numbers of implications kept during the analysis at each program point. For the equality domain, we give the average and the maximum numbers of equalities (in implications) propagated by the analysis at each program point. A timeout denotes an analysis time that exceeds 10 minutes.

The introduction of the difference operation has little impact on the performance analysis for the initialization domain. In contrast, it greatly improves the equality analysis, as it removes many equalities from the implications, and all the operations on equality sets depend

<pre> 1 if (i >= 0 && i < 10) x = 1; else x = 0; 5 [...] if (x==1) /*@ assert 0 <= i < 10; */ a[i] = 42; </pre>	<pre> 1 p = &x; while (n > 0) { /*@ assert p != 0; */ x = *p + x; n--; if (!(n > 0)) p = 0; } </pre>
---	--

Figure 9.11: Examples of disjunctions in the literature

on their size. Not surprisingly, the benefits of the difference operation depend on the underlying abstract domain, as its aim is to reduce the size of the abstract values.

The application of a lighter transfer function for the implications speeds up substantially the analysis for both domains. Not only this new function is itself faster than the original one, but it also leads to an important decrease of superfluous implications. Indeed, the original transfer function alters systematically all implications, and thus the shared subparts of two **CI**-pairs are minimized after a disjunction. With the lighter transfer function, most of the implications collected before a split in the control-flow are propagated in the branches without any change, and are kept unchanged in the join. The combination of these implications is then avoided, thanks to the optimization of the join presented at the end of Section 9.3.3.

9.7.5 Experiments on Examples from the Literature

We have successfully applied our predicated analyses, instantiated with the domain of equalities, to various examples of the literature [FJMo5; San+o6b; MS14; HHP13] —starting with the motivating example introduced in Figure 9.1. To analyze it, we simply modeled the open function as a random assignment to 1 or -1 (corresponding to a successful or failed call, respectively), and replaced the calls to `close` by an assertion requiring the file descriptor to be 1. The implications gathered along the analysis link `flag1` and `flag2` to the value of `fd1` and `fd2`, and the `close` assertions are directly proved.

Two other interesting examples are presented in Figure 9.11. The properties required for the program to be correct are given as ACSL assertions. The left one requires the variable i to be within the bounds of the array a at line 8, which is effectively ensured by the condition $x == 1$. This pattern —storing a predicate within a boolean, which is tested later— is actually quite frequent. Disjunctive domains handle this code naturally, since they propagate two complete separate states after the disjunction, while our analysis is guided by the meaning of the implications. At line 6, we have no implication of the form $\langle x = c \rightarrow _ \rangle$; however, we have $\langle \neg (0 \leq i < 10) \rightarrow x = 0 \rangle$, which be-

comes $\langle \neg (0 \leq i < 10) \rightarrow \perp \rangle$ at line 7. The predicate $0 \leq i < 10$ can then be added to the context, as stated in Section 9.4.3. This is sufficient to validate the assertion.

Finally, disjunctive domains may distinguish loop iterations, by propagating one abstract state for each iteration. Without specific predicates able to label each iteration, our framework cannot offer the same expressiveness. Still, we can sometimes convey relevant information through a loop. The example on the right of Figure 9.11 shows a loop in which a pointer p is dereferenced and then freed (set to 0) in its last iteration. Our predicated analyses infer $\langle \neg (n > 0) \rightarrow p = 0 \rangle$ and $\langle n > 0 \rightarrow p = \&x \rangle$, thus ensuring the validity of dereferencing p in the loop, where the context is $n > 0$.

9.8 CONCLUSION

We have described a generic framework to enhance the precision of standard dataflow analyses. This framework constructs a derived predicated analysis able to mitigate information loss at junction points of the control-flow graph, by retaining the conditional values about each branch. Our analysis strives to minimize redundant information processing due to these disjunctions. Experimental tests led through a dedicated plugin of FRAMA-C, applied to generated C code, showed that a predicated analysis over simple domains can significantly improve the results of prior analyses.

9.8.0.1 Future works

The literals of our predicates are expressions that we currently consider as uninterpreted. In order to improve our analysis, we intend to give some meaning to the operators in these expressions and to extend the logical implication between guards accordingly. In particular, we could handle successive conditions on distinct but related expressions, such as $(x \geq 0) ?$ and $(x \geq 2) ?$. The difficulty lies in finding an equational theory for entailments \vdash that would be expressive enough, but not too costly.

Another interesting extension would be to preserve more information when encountering invertible assignments such as $x := x + 1$. Currently, all information about x is lost in the context and the guards afterwards. This requires some care to avoid producing an infinite number of new predicates, which would endanger the convergence of the analysis.

Moreover, prior syntactic analyses or heuristics could help to select relevant predicates for the contexts, which would no longer be extended at each test statement. This would avoid maintaining implication guards that will never be useful again later in the program. Likewise, we could use heuristics to define variable packing strategies, in order to limit the abstract states themselves. The boolean

partitioning used in Astrée [Ber+10] keeps information only for some syntactically well-chosen variables. We could do the same, by keeping in the state under a guard p only the variables that are related to p in the program.

Finally, it would be worthwhile to apply our predicated analysis over more complex abstract domains.

Part VI

CONCLUSION

10.1 SUMMARY

In this thesis, we have proposed a new framework for the combination of multiple domains in the abstract interpretation theory. Its core concept is the structuring of the abstract semantics by following the usual distinction between expressions and statements. This can be achieved by a convenient architecture where abstractions are separated in two layers: value abstractions, in charge of the expression semantics, and state abstractions—or abstract domains—in charge of the statement semantics. This design leads to a natural communication system where the abstract domains, when interpreting a statement, interact and exchange information through value abstractions, that express properties about expressions. While the values form the communication interface between domains, they are also standard elements of the abstract interpretation framework. The communication system is thus embedded in the abstract semantics, and the usual tools of abstract interpretation apply naturally to value abstractions. For instance, different kinds of value abstractions can be composed through the existing methods of combination of abstractions. This thesis has explored the possibilities offered by this framework, and we have described strategies to compute precise value abstractions from the information inferred by abstract domains. The architecture also features a direct collaboration for the emission of alarms that report the possible errors of a program.

This design has been implemented within [EVA](#), the new version of the abstract interpreter provided by the FRAMA-C platform. [EVA](#) is a major evolution of the former abstract interpreter of FRAMA-C, named [VALUE](#). Unlike [VALUE](#), [EVA](#) enjoys a modular and extensible architecture designed to ease the introduction of new abstractions. Thanks to this work, five new domains from the literature have been implemented in the last year, enhancing the scope of the analyzer. Considerable efforts have also been devoted to preserve the efficiency of the analyzer. To maintain some crucial optimizations relying on a specific abstraction, we have developed a [GADT](#) structure allowing direct and efficient interactions with the components of a generic product of OCaml data types (Section [3.3](#)). We have also defined a formal semantics for the value abstractions called garbled mixes, that soundly represent the results of integer arithmetic applied to pointer values (Section [5.3](#)). In the end, [EVA](#) is now an accomplished project:

it is already used in industrial case studies, and has definitely superseded [VALUE](#) in the FRAMA-C framework.

10.2 FUTURE WORKS IN EVA

This section outlines diverse developments that are planned to improve [EVA](#) in the short to medium term. They derive from the work done in this thesis or from the state of the art, and aim mainly to facilitate the implementation of even more domains.

SIMPLIFY THE API The foremost motivation for the development of [EVA](#) was to enable and to ease the introduction of new abstractions in the analyzer. The modular architecture and the communication system of [EVA](#) meet these goals. Nonetheless, the programming interface of abstract domains is still relatively complicated; some parts are a remnant of the former analyzer, such as the functions for the initialization of the initial state of the analysis—that have not been mentioned in this thesis. Much work remains to be done towards the simplification, the standardization and the documentation of the [API](#) of [EVA](#). Moreover, we would like to provide support tools to build complete domains from simpler abstractions—such as domains that do not interact with others, or simple observers that do not participate in the computations. We also intend to develop a simpler functor to automatically manage the memory model of C for purely numerical domains (the memory functor of the [CVALUE](#) domain is far too complex to play this role). This functor could follow the preliminary work undertaken with the binding of the [APRON](#) domains.

EXTEND THE VALUE ABSTRACTIONS All means of communication between abstract domains involve value abstractions. Even if the reduced product of value abstractions currently available in [EVA](#) already establishes a rich and expressive interface, new value abstractions may enable further interactions. We draw here a few avenues worth exploring to extend value abstractions.

- In general, intervals cannot precisely represent the possible values of an unsigned integer variable in case of arithmetic overflow (which is not an undefined behavior for unsigned integers in the C standard). At the end of Listing [10.1](#), the variable x can be 0 or $2^n - 1$ (where n is the size in bits of an integer). The most precise interval that contains these two values is $[0..2^n - 1]$, which is overly imprecise. A dedicated value abstraction may circumvent this loss of precision.
- The disjunctive union of intervals is a well-known refinement of intervals [[GC10a](#)] that enables the precise approximation of

Listing 10.1: Unsigned integer overflow in C

```

1 unsigned int x = 0;
  if (c) x--;

```

non-convex sets of values. To remain scalable, some strategies must be found to bound the number of disjuncts.

- Some relational domains rely on the linear approximation of expressions [Mino6b], that is the simplification of arbitrary expressions into affine forms with interval coefficients. The linearization of an expression could be easily encoded as a value abstraction, in a reduced product with intervals to extract the coefficients that approximate nonlinear expressions.

COOPERATIVELY EVALUATE LOGICAL ASSERTIONS The FRAMA-C platform embeds the [ACSL](#) language for the formal specification of a program through logical annotations. These annotations can be understood by its diverse plugins—and by the abstract domains of [EVA](#). [ACSL](#) annotations can be written by the user or by another plugin. Then, the domains of [EVA](#) can prove some annotations (if they are logically implied by the inferred abstract state), or use them to reduce their states (and rely on other plugins to prove them). For the time being, only the main `CVALUE` domain understands and processes the logical assertions. The cooperation between the domains of [EVA](#) is currently limited to the interpretation of C statements—through the shared evaluation of C expressions. The same collaborative mechanism should also be applied to the interpretation of [ACSL](#) assertions, by a shared evaluation of the logical terms. However, this requires some weighty works, as the [ACSL AST](#) is much larger than the C [AST](#).

IMPROVE THE STATE PARTITIONING To achieve further precision, [EVA](#) uses an automatic partitioning of abstract domains and infers a disjunction of states at each point of a program. To remain scalable, the maximal number of disjuncts is bounded by a parameter of the analysis. Apart from this parameter, the strategy for the choice of the state partitioning is naive, cannot be adjusted for specific needs, and rests upon optimization heuristics tailored to the main `CVALUE` domain. Further development should allow the user to specify criteria and directives for both the disjunction and the junction of states during the analysis. Moreover, trace partitioning [MR05] is a technique that relates each state with a trace abstraction of the execution paths it represents. Then, choosing a partition of the traces (through heuristics or user commands) can guide the disjunction of states. A form of trace partitioning could greatly improve the precision and the user-friendliness of [EVA](#).

DEVELOP AND HONE NEW ABSTRACTIONS WITHIN EVA Finally, we should carry on reaping the rewards of this thesis, by developing and improving even further the domains of [EVA](#). On the one hand, the new abstract domains that have been implemented last year remain mostly experimental; they need to be strengthened to reach the same level of maturity as the inherited `CVALUE` domain. On the other hand, new classes of worthwhile domains can be contemplated to further enrich the abstract interpreter. For instance, abstract domains for shape may be required to precisely analyze programs including linked and dynamically allocated data structures. Moreover, it is difficult to exactly foresee the specific needs of new domains before trying to implement them. Confronting our framework to a wide range of abstract domains will bring its strengths and weaknesses to light. Such experiments should guide our further developments of the [EVA](#) internal architecture.

10.3 LONG-TERM PERSPECTIVES

This section presents some ideas of future research directions that we deem worthwhile and promising, to go along with or beyond the concepts proposed in this thesis.

EXTENSION OF THE COMMUNICATION SYSTEM Our successful experiments on various abstract domains have convinced us of the suitability and the merits of a communication through value abstractions. Nonetheless, this communication system could be extended to create further opportunities for domains to interact, leading to more accurate analyses. We sketch here some ideas for further improvements.

Firstly, the interactions are for now limited to the interpretation of statements. The domains could also benefit from interactions during join operations, where some precision is usually lost. At join points, there are no natural expressions to evaluate. Nevertheless, any expression can still be evaluated (or reduced by backward propagation) at the request of domains that want to acquire (or send) information through value abstractions. This would enable nearly the same reductions between abstract states at the merge points of the [CFG](#) as those on regular statements. More generally, we could extend the possibilities of interactions by providing the domains with a greater access to the evaluation functionalities. In our design, a domain can send information to the others by asserting a value for an expression—the value is then backward propagated through the expression. Currently, a domain can initiate a backward propagation only when a value abstraction is reduced (the new backward propagation must then be a consequence of the reduction). However, some domains

may want to initiate backward propagations on other circumstances that need to be identified.

Secondly, when answering a query, a domain is unaware of the context of the operation, such as the program point, the expression being evaluated, or the abstractions computed so far by other domains. This information may assist a costly domain to choose the right trade-off between precision and efficiency on a case-by-case basis. For instance, a domain could choose to perform more complex but more accurate computation at some specific points of interest, or when the abstractions provided by other domains remain overly imprecise, or where an alarm can be avoided.

Finally, value abstractions can only express properties about the expressions of the language. While this expressivity seems perfectly suitable for a cooperative interpretation of statements one by one, a more advanced communication system may involve properties about the effect of a block of statements, a loop, or a whole function. However, this would require the design of brand new kinds of interactions altogether.

EFFICIENT COMBINATION OF DIFFERENT DOMAINS Regardless of the interactions established between abstractions, the combination of very different domains may cause serious efficiency issues. First, the high complexity of some domains impedes their scalability on large programs, while the reasoning they conduct can be highly valuable in some settings. A modular abstract interpreter should enable the activation of such domains on only parts of the analyzed program, according to automatic heuristics or to criteria specified by the user. Secondly, different domains may have different convergence speeds, depending on their lattice structure and the implementation of the widening operator. Even without any inter-reduction between abstract states, the convergence speed of a domain product is that of the slowest domain. This implies that the faster domains perform unnecessary computations through the last iterations needed to reach the fixpoint of the slowest domain. Interplay between domains may also slow down the convergence, and makes the optimizations of the fixpoint computation even harder, since the states of a domain may be reduced after reaching a first fixpoint. While this thesis has tackled the challenge of designing a *modular* and *practical* communication system between various abstractions, progress needs to be strengthened toward the *efficient* combination of arbitrary domains.

SUMMARY OF FUNCTION ANALYSES The strong efficiency of the CVALUE domain, inherited from the [VALUE](#) plugin, is partly due to an automatic summarization mechanism that speeds up analyses [[Yak15](#)]. This cache mechanism has been specifically tailored to this domain, and its soundness relies on the absence of relational properties. It

now needs to be extended to arbitrary domains while remaining cost-efficient. In particular, the use of relational properties significantly complicates the computation of a concise summary, as the set of variables that serves for the analysis of a function cannot be easily determined.

OTHER FORMAL VERIFICATION TECHNIQUES This thesis is dedicated to the formal verification of programs through abstract interpretation, by uniting the strengths of manifold abstract domains. An ambitious research direction would be to unite the strengths of different techniques of formal verification. FRAMA-C is a platform that already gathers a set of interoperable tools based on various verification technologies. Although these different tools can be applied to the same program, each one works mostly independently from the others. However, they could certainly benefit from closer interactions. In particular, abstract interpretation and weakest precondition calculus can be intertwined to achieve the automation of a program proof or a better precision. On the one hand, abstract interpretation has already been successfully applied to infer the loop invariants required to the computation of weakest preconditions [Bar+05]. On the other hand, abstract interpretation based analyses can be improved by resorting to decision procedures and SMT solvers [HMM12].

Part VII

APPENDIX

NOTATIONS SUMMARY

Color notation: **Concrete** vs **Abstract**.

VARIABLE	variables	$x \in \mathcal{X}$	4.2.1
	memory layouts	$\theta \in \Theta_P$	4.2.2
C VALUES	C values of τ type	$c \in \bar{\mathbb{V}}_\tau$	4.2.1
	... including pointer values	$(\&x, i) \in \bar{\mathbb{V}}_{\text{ptr}}$	4.2.1
	all scalar C values	$c \in \bar{\mathbb{V}}$	4.2.1
	byte values	$\bar{\mathbb{V}} \triangleq \{0, \dots, 255\}$	4.2.2
	undeterminate values	uninit , none	4.2.2.2
	interpretation functions	$\phi_\tau : (\bar{\mathbb{V}}_{\text{bytes}})^{\text{sizeof}(\tau)} \rightarrow \bar{\mathbb{V}}_\tau$	4.2.2
VALUES	concrete values as functions from layouts to C values	$V \in \mathcal{V} : \Theta_P \rightarrow \bar{\mathbb{V}}$	4.3.3
	error value	Ω	4.2.4
	value abstraction	$v \in \mathbb{V}^\#$	5.2.1
	value concretization	$\gamma_{\mathcal{V}} : \mathbb{V}^\# \rightarrow \mathcal{P}(\mathcal{V})$	5.2.1
	value lattice structure	$\gamma_{\mathcal{V}} : \mathbb{V}^\# \rightarrow \mathcal{P}(\mathcal{V})$	5.2.2
LOCATIONS	valid location of type τ ¹	\mathcal{L}_τ	4.2.3.1
	abstract location	$\mathbb{L}^\#$	5.2.6
	location concretization	$\gamma_{\mathbb{L}} : \mathbb{L}^\# \rightarrow \mathcal{P}(\mathcal{L})$	5.2.6
	location lattice structure	$\sqsubseteq_{\mathbb{L}}, \sqcup_{\mathbb{L}}, \sqcap_{\mathbb{L}}, \top_{\mathbb{L}}$	5.2.6
ALARMS	possible alarms for an expression	alarms (e)	5.1.3
	alarm map	$\mathbf{A} \in \mathbb{A}$	5.1.4
	alarm map soundness	$\models \mathbf{A}$	5.1.4
	alarm map lattice	$\sqsubseteq_{\mathbb{A}}, \sqcup_{\mathbb{A}}, \sqcap_{\mathbb{A}}, \top_{\mathbb{A}}$	5.1.4

¹ always independent of memory layouts

OPERATORS	n-ary operator	\diamond	4.2.1
	application of \diamond	$[\diamond] : \overline{\mathcal{V}}^n \rightarrow \overline{\mathcal{V}}$	4.2.4
	concrete semantics of \diamond	$\llbracket \diamond \rrbracket^\Theta : (\mathcal{V})^n \rightarrow \mathcal{V}$	4.3.3
	forward abstract semantics	$F_\diamond^\#$	5.2.5
	backward abstract semantics	$B_\diamond^\#$	5.2.5
STATES	memories	$\mathbf{m} \in \mathfrak{M}$	4.2.3.1
	concrete states	$S \in \mathcal{S} = \mathfrak{M}^{\Theta_P}$	4.3.2
	state abstractions (abstract domain)	$D \in \mathbb{D}$	7
	domain concretization	$\gamma_{\mathbb{D}} : \mathbb{D} \rightarrow \mathcal{P}(\mathcal{S})$	7
	domain lattice	$\sqsubseteq_{\mathbb{D}}, \sqcup_{\mathbb{D}}, \sqcap_{\mathbb{D}}, \top_{\mathbb{D}}$	7
	forward dereference semantics	$F_{*\tau}^\#$	7.1.1
	backward dereference semantics	$B_{*\tau}^\#$	7.2.1
	forward query on arbitrary expression	$F_{\mathbb{D}}^\#$	7.1.2
	backward query on arbitrary expression	$B_{\mathbb{D}}^\#$	7.2.2
VALUATIONS	valuations	$\mathcal{E} \in \mathbb{E}$	6.1.3
	valuation concretization	$\gamma_{\mathbb{E}} : \mathbb{E} \rightarrow \mathcal{P}(\mathcal{S})$	6.1.3
	valuation lattice	$\sqsubseteq_{\mathbb{E}}, \sqcup_{\mathbb{E}}, \sqcap_{\mathbb{E}}, \top_{\mathbb{E}}$	6.1.3

PROOFS

LEMMA 14

Lemma. *Given a program P with a set \mathcal{X} of variables and two memory layouts θ and θ' in Θ_P , axiom 2 ensures the existence of a series of valid memory layouts $\theta_0, \dots, \theta_n$ such that $\theta_0 = \theta$, $\theta_n = \theta'$ and for all i , θ_{i-1} and θ_i differ on at most one variable.*

$$\forall i \in \{1, \dots, n\}, \exists x \in \mathcal{X}, \forall y \in \mathcal{X}, y \neq x \Rightarrow \theta_{i-1}(y) = \theta_i(y)$$

Proof. By induction on the number of variables. For one variable, the result is trivial. We assume the lemma true for any set of $n-1$ variables, and we consider a set \mathcal{X} of n variables.

We fix a special memory layout θ' and we prove the existence of a series of *adequate* layouts from any layout θ to θ' —adequate meaning that two successive layouts differ on at most one variable. Then, for any layouts θ and θ'' , two series of adequate layouts from θ to θ' and from θ'' to θ' can be combined into a series of adequate layouts from θ to θ'' . We name the variables of $\mathcal{X} = \{y_1, \dots, y_n\}$, and fix θ' as:

$$\theta'(y_k) = 2^{\text{sizeof}(\text{ptr})} - 1 - \sum_{j \in \{k, \dots, n\}} (\text{sizeof}(y_j) + 1)$$

This function from Θ_P to \mathbb{N} is a memory layout according to definition 21:

1. Strictly positive addresses by axiom 2:

$$\begin{aligned} \forall k, \theta'(y_k) &= 2^{\text{sizeof}(\text{ptr})} - 1 - \sum_{j \in \{k, \dots, n\}} (\text{sizeof}(y_j) + 1) \\ &\geq 2^{\text{sizeof}(\text{ptr})} - 1 - \sum_{j \in \{1, \dots, n\}} (\text{sizeof}(y_j) + 1) \\ &> \max_{x \in \mathcal{X}} (\text{sizeof}(y)) > 0 \end{aligned}$$

2. Non-aliasing:

$$\begin{aligned} \forall k > k', \theta'(y_k) - \theta'(y_{k'}) &= \sum_{j \in \{k, \dots, k'-1\}} (\text{sizeof}(y_j) + 1) \\ &\geq \text{sizeof}(y_k) + 1 > \text{sizeof}(y_{k'}) \end{aligned}$$

3. Fits in memory:

$$\begin{aligned} \forall k, \theta'(y_k) &= 2^{\text{sizeof}(\text{ptr})} - 1 - \sum_{j \in \{k, \dots, n\}} (\text{sizeof}(y_j) + 1) \\ &< 2^{\text{sizeof}(\text{ptr})} - \text{sizeof}(y_k) \end{aligned}$$

We now consider an arbitrary memory layouts θ . We construct the series of layouts θ_0 to θ_k . The idea is to shift one variable at a time in θ until all variables are adjacent. Then, axiom 2 ensures that

We rename the variables of \mathcal{X} as x_1, \dots, x_n such that

$$j \leq k \Rightarrow \theta(x_j) \leq \theta(x_k)$$

We then define the functions θ_0 to θ_n as follows:

$$\begin{aligned} \theta_0 &= \theta \\ \forall i \in \{1, \dots, n\}, \quad &\begin{cases} \forall k \neq i, \theta_i(x_k) = \theta_{i-1}(x_k) \\ \theta_i(x_i) = \sum_{j \in \{1, \dots, i-1\}} (\text{sizeof}(x_j) + 1) + 1 \end{cases} \end{aligned}$$

Let us prove inductively that for all $i \in \{0, \dots, n\}$, θ_i is a valid layout, and that the variables x_1 to x_n are still ordered in θ_i .

$$j \leq k \Rightarrow \theta_i(x_j) \leq \theta_i(x_k)$$

It is true for θ_0 by hypothesis. We assume it is also true for θ_{i-1} , and prove it for θ_i . As θ_i is equal to θ_{i-1} except on x_i , we only have to consider this variable.

1. Strictly positive address: $\theta_i(x_i) > 0$.
2. Non-aliasing with “smaller” variables x_k with $k < i$:

$$\begin{aligned} \theta_i(x_{i-1}) &= \theta_{i-1}(x_{i-1}) = \sum_{j \in \{1, \dots, i-2\}} (\text{sizeof}(x_j) + 1) + 1 \\ \Rightarrow \theta_i(x_i) - \theta_i(x_{i-1}) &= \text{sizeof}(x_{i-1}) + 1 > \text{sizeof}(x_{i-1}) \end{aligned}$$

And for any variable x_k with $k < i - 1$, as θ_{i-1} is valid:

$$\theta_{i-1}(x_{i-1}) - \theta_{i-1}(x_k) > \text{sizeof}(x_k)$$

And:

$$\begin{aligned} \theta_i(x_i) - \theta_i(x_k) &= \theta_i(x_i) - \theta_{i-1}(x_k) \\ &= (\theta_i(x_i) - \theta_{i-1}(x_{i-1})) + (\theta_{i-1}(x_{i-1}) - \theta_{i-1}(x_k)) \\ &> \text{sizeof}(x_{i-1}) + \text{sizeof}(x_k) \\ &> \text{sizeof}(x_k) \end{aligned}$$

3. Non-aliasing with “greater” variables x_k with $k > i$:
By inductive hypothesis, $\theta_{i-1}(x_{i-1}) \leq \theta_{i-1}(x_i)$.
As θ_{i-1} is a memory layout according to definition 21:

$$\begin{aligned} \theta_{i-1}(x_i) - \theta_{i-1}(x_{i-1}) &\geq \text{sizeof}(x_{i-1}) + 1 \\ \Rightarrow \theta_{i-1}(x_i) &\geq \theta_{i-1}(x_{i-1}) + \text{sizeof}(x_{i-1}) + 1 \\ &\geq \sum_{j \in \{1, \dots, i-1\}} (\text{sizeof}(x_j) + 1) + 1 \\ &\geq \theta_i(x_i) \end{aligned}$$

For $k > i$, $\theta_i(x_k) = \theta_{i-1}(x_k)$ and:

$$\begin{aligned}\theta_i(x_k) - \theta_i(x_i) &\geq \theta_{i-1}(x_k) - \theta_{i-1}(x_i) \\ &\geq \text{sizeof}(x_k)\end{aligned}$$

(Still by definition 21)

4. The address fits in memory thanks to axiom 2:

$$\begin{aligned}\theta_i(x_i) &= \sum_{j \in \{1, \dots, i-1\}} (\text{sizeof}(x_j) + 1) + 1 \\ &\leq \sum_{x \in \mathcal{X}} (\text{sizeof}(x) + 1) + 1 \\ &< 2^{\text{sizeof}(\text{ptr})} - \max_{x \in \mathcal{X}} (\text{sizeof}(x)) \\ &\leq 2^{\text{sizeof}(\text{ptr})} - \text{sizeof}(x_i)\end{aligned}$$

5. Point 2 and 3 have also proved the invariant

$$\begin{cases} k < i \Rightarrow \theta_i(x_i) < \theta_i(x_k) \\ k > i \Rightarrow \theta_i(x_i) > \theta_i(x_k) \end{cases}$$

Thus, the series of functions θ_0 to θ_n are memory layouts, according to definition 21. By definition, for any $i \in \{1, \dots, n\}$, θ_i and θ_{i-1} are equal except on x_i .

We finally consider the function θ_{n+1} defined as:

$$\theta_{n+1}(x) = \begin{cases} \theta_n(x) & \text{if } x \neq y_n \\ 2^{\text{sizeof}(\text{ptr})} - (\text{sizeof}(y_n) + 1) & \text{if } x = y_n \end{cases}$$

We can note that $\theta_{n+1}(y_n) = \theta'(y_n)$. Let us prove that this last function θ_{n+1} is also a memory layout. As θ_n is a memory layout, we only consider the variable y_n on which θ_{n+1} differs.

As $\theta_{n+1}(y_n) = \theta'(y_n)$ or by axiom 2, we directly have:

$$0 < \theta_{n+1}(y_n) < 2^{\text{sizeof}(\text{ptr})} - \text{sizeof}(y_n)$$

Let x_i be a variable different from y_n . Then:

$$\begin{aligned}\theta_n(x_i) &= \sum_{j \in \{1, \dots, i-1\}} (\text{sizeof}(x_j) + 1) + 1 \\ &\leq \sum_{j \neq i} (\text{sizeof}(x_j) + 1) + 1 \\ &< 2^{\text{sizeof}(\text{ptr})} - \max_{y \in \mathcal{X}} (\text{sizeof}(y)) + \text{sizeof}(x_i)\end{aligned}$$

$$\begin{aligned}\theta_{n+1}(y_n) - \theta_{n+1}(x_i) &= 2^{\text{sizeof}(\text{ptr})} - \text{sizeof}(y_n) - \theta_n(x_i) \\ &\geq 2^{\text{sizeof}(\text{ptr})} - \max_{y \in \mathcal{X}} (\text{sizeof}(y)) - \theta_n(x_i) \\ &> \text{sizeof}(x_i)\end{aligned}$$

SUMMARY: We have a series of memory layouts θ_0 to θ_{n+1} such that $\theta_0 = \theta$, $\theta_{n+1}(y_n) = \theta'(y_n)$ and for all $i \in \{0, \dots, n\}$, θ_i and θ_{i+1} differ on at most one variable (this is immediate by their definitions).

Let $M = 2^{\text{sizeof}(\text{ptr})} - \text{sizeof}(y_n) - 1$. We now consider the restriction of θ_{n+1} and θ' from the set of variables $\mathcal{X} \setminus \{y_n\} = \{y_1, \dots, y_{n-1}\}$ to the integer addresses strictly between 0 and M . They are valid memory layouts on a set of $n - 1$ variables in a space addressing verifying axiom 2. Indeed, we have:

$$\begin{aligned}
 M &= 2^{\text{sizeof}(\text{ptr})} - \text{sizeof}(y_n) - 1 \\
 &> \sum_{y \in \mathcal{X}} (\text{sizeof}(y) + 1) + 1 + \max_{y \in \mathcal{X}} (\text{sizeof}(y)) - \text{sizeof}(y_n) - 1 \\
 &> \sum_{y \in \mathcal{X} \setminus \{y_n\}} (\text{sizeof}(y) + 1) + 1 + \max_{y \in \mathcal{X} \setminus \{y_n\}} (\text{sizeof}(y))
 \end{aligned}$$

Thus, the inductive hypothesis applies: there exists a series of adequate memory layouts from θ_{n+1} to θ' , restricted to the variables $\mathcal{X} \setminus \{y_n\}$ and the addressing space $\{1, \dots, M - 1\}$. We can complete each memory layout θ_i of this series by $\theta_i(y_n) = M$, and obtain a series of adequate memory layouts from θ_{n+1} to θ' without restriction.

We have finally constructed a series of adequate memory layouts from an arbitrary layout θ to the fixed memory layout θ' . \square

DEVELOPMENT FILES

File	Main functionalities	Ref
<code>alarmset.mli</code>	Maps of alarms, embodying the undesirable behaviors tracked down by EVA	5.1
<code>eval.mli</code>	Types heavily used in EVA (including bottom and alarm monads, the valuations and some types about assignments and function calls)	5.1.5 6.1.3
<code>value_parameters</code>	Options of the analyzer	
<code>engine/</code>	Generic abstract interpreter	3.2.1
<code>evaluation</code>	Evaluation of expressions	6
<code>non_linear_evaluation</code>	Subdivision of an evaluation	6.2.4
<code>transfer_stmt</code>	Interpretation of statements	3.2.1
<code>transfer_logic</code>	Interpretation of logical assertion — incomplete	
<code>partitioning</code>	Disjunctive powerset of an abstract domain	3.2.1
<code>partitioned_dataflow</code>	Dataflow on a disjunctive powerset domain	3.2.1
<code>mem_exec</code>	Cache for the dataflow of a function body	3.2.1
<code>initialization</code>	Generation of the abstract state at the entry point of a program	3.2.1
<code>abstractions</code>	Instantiation of the abstractions (according to the options of the analyzer)	3.2.3
<code>compute_functions</code>	Analysis of an entire function and beginning of the analysis (<i>Computation</i> functor on Figure 3.2)	3.2.1
<code>values/</code>	Abstractions of values	5
<code>abstract_value</code>	Signature of value abstractions	5.2
<code>abstract_locations</code>	Signature of location abstractions	5.2.6
<code>cvalue_forward</code>	Forward abstract transformers of cvalues ¹	5.3.3
<code>cvalue_backward</code>	Backward abstract transformers of cvalues	5.3.4

¹ The implementation of cvalues is in `src/plugins/value_types/cvalue.ml`

offsm_values	Bitwise value abstractions	
main_values	Value abstractions currently provided by EVA (including cvalues)	5.3
main_locations	Location abstractions currently provided by EVA	5.3
value_product	Reduced product of value abstractions	3.2.2
domains/	Abstract domains	iv
abstract_domain	Signature of abstract domains	7
domain_builder	Automatic generation of some domain features	
domain_lift	Lifts a domain to another value abstraction	3.2.2
domain_product	Combines two domains working on the same value abstractions	3.2.2
cvalue/	Legacy memory domain	8.1
apron/	Binding with the numerical abstract domains of the APRON library	8.3.1
equality/	Symbolic equality domain	8.2
gauges/	Gauges domain	8.3.3
offsm_domain	Bitwise abstract domain	8.3.4
symbolic_locs	Symbolic location domain	8.3.2
utils/	Utility files shared between VALUE and EVA	
structure	Structure of a datatype	3.3
legacy/	The legacy abstract interpreter VALUE	
gui_files/	The graphical user interface of VALUE/EVA	

LIST OF FIGURES

Figure 2.1	Syntax of the Toy language	20
Figure 2.2	Denotational semantics of Toy	21
Figure 2.3	Euclidean algorithm	23
Figure 2.4	Lattices	30
Figure 2.5	Need for interactions between abstract domains	38
Figure 2.6	Need for interactions during statements interpretation	41
Figure 3.1	Overall layers of EVA	52
Figure 3.2	Architecture of the analyzer	56
Figure 4.1	Syntax of the <code>clike</code> language	81
Figure 4.2	From C to <code>clike</code>	83
Figure 4.3	Interpretation functions and memory layouts .	84
Figure 4.4	Evaluation of expressions and addresses	88
Figure 4.5	Jump between variables via integer arithmetic	91
Figure 4.6	Pointer union	92
Figure 4.7	Pointer arithmetic through integers	92
Figure 4.8	Concrete semantics of expressions and addresses	93
Figure 4.9	Concrete semantics of statements	95
Figure 5.1	Statuses and alarms	107
Figure 5.2	OCaml types for the alarms and the bottom case	111
Figure 5.3	Value abstraction of some expressions	113
Figure 5.4	Interface of the abstract operators on values . .	118
Figure 5.5	Lattice structure of the <code>cvalue</code> abstractions . .	128
Figure 6.1	The evaluation functor	142
Figure 6.2	Backward propagation after alarm emission . .	159
Figure 6.3	Backward propagation on a conditional statement	161
Figure 6.4	Backward propagation from an abstract domain	163
Figure 6.5	Forward propagation after a backward reduction	165
Figure 6.6	Slow convergence of interval propagations . .	166
Figure 7.1	Forward collaboration between domains	182
Figure 7.2	Collaboration between domains on arbitrary expressions	186
Figure 7.3	Interaction through the oracle	191
Figure 7.4	Backward propagation on an array access . . .	193
Figure 7.5	Backward propagation on a pointer dereference	194
Figure 7.6	Triggering new reductions	196
Figure 7.7	Using the oracle or the reducer	197
Figure 7.8	Interpreting a statement	202
Figure 8.1	Assignment modifying the address	225

Figure 8.2	Experimental results on the new abstract domains	232
Figure 9.1	Example of interleaved conditionals	238
Figure 9.2	Syntax of our language	239
Figure 9.3	Concrete and abstract semantics	240
Figure 9.4	Example of an analysis using a predicated interval analysis	247
Figure 9.5	Definition of the abstract semantics $\llbracket \cdot \rrbracket_{\text{pred}}^{\#}$. .	248
Figure 9.6	Extended assume to predicates in disjunctive normal form.	249
Figure 9.7	Analysis of Figure 9.4 with factorization. . . .	252
Figure 9.8	Abstract semantics for the equality domain . .	262
Figure 9.9	Experimental results	263
Figure 9.10	Effectiveness of the optimizations	265
Figure 9.11	Examples of disjunctions in the literature . . .	266

LIST OF TABLES

Table 2.1	Numerical abstract domains in the literature .	35
Table 8.1	Comparisons between VALUE and EVA	213
Table 8.2	Comparisons between different evaluation strategies	215
Table 8.3	Experimental results of the equality domain .	229

LISTINGS

Listing 3.1	Combination of Datatypes	61
Listing 3.2	Keys and interface	62
Listing 3.3	Implementing keys with GADT	63
Listing 3.4	Using extensible types to implement polymorphic keys	63
Listing 3.5	Exporting keys	64
Listing 3.6	Naïve implementation of the External signature	65
Listing 3.7	Structure of a Datatype	66
Listing 3.8	Dependent Maps	67
Listing 3.9	Accessors generation	68
Listing 4.1	Detection of undefined behaviors	77
Listing 4.2	Object representation of pointers	78
Listing 5.1	Reducing abstract location	120

Listing 5.2	Basic pointer abstraction representing integers	125
Listing 6.1	Implementation of a simplified evaluator . . .	156
Listing 6.2	Avoiding redundant alarms	172
Listing 6.3	Second forward propagation on conditional ex- pression	173
Listing 6.4	Subdivision of evaluation	174
Listing 7.1	Signature of domain queries	180
Listing 7.2	Assignment through a pointer	184
Listing 7.3	Implementation of an oracle in the evaluator .	188
Listing 7.4	Signature of domain backward propagators . .	191
Listing 10.1	Unsigned integer overflow in C	273

ACRONYMS

ACSL [ANSI/ISO C Specification Language](#)

ANSI American National Standards Institute

API Application Programming Interface

AST Abstract Syntax Tree

CIL C Intermediate Language

CFG Control-Flow Graph

GADT Generalized Algebraic Data Type

GUI Graphical User Interface

ISO International Organization for Standardization

SCADE Safety Critical Application Development Environment

SMT Satisfiability Modulo Theories

EVA Evolved Value Analysis

VALUE the legacy Value Analysis of Frama-C

CI Context and Implication map

BIBLIOGRAPHY

- [ACSL] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, Version 1.12*. Tech. rep. <http://frama-c.com/download/acsl.pdf>. CEA LIST and INRIA, 2016 (cit. on pp. 75, 102, 103).
- [Aym39] Marcel Aymé. *Les contes du chat perché*. Gallimard, 1939 (cit. on p. x).
- [Bag+05] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. “Precise widening operators for convex polyhedra”. In: *Sci. Comput. Program.* 58.1-2 (2005), pp. 28–56 (cit. on p. 33).
- [Bar+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387 (cit. on p. 276).
- [BBY14] Sandrine Blazy, David Bühler, and Boris Yakobowski. “Improving Static Analyses of C Programs with Conditional Predicates”. In: *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*. Ed. by Frédéric Lang and Francesco Flammini. Vol. 8718. Lecture Notes in Computer Science. Springer, 2014, pp. 140–154 (cit. on p. 14).
- [BBY16] Sandrine Blazy, David Bühler, and Boris Yakobowski. “Improving static analyses of C programs with conditional predicates”. In: *Sci. Comput. Program.* 118 (2016), pp. 77–95 (cit. on pp. 14, 237).
- [BBY17] Sandrine Blazy, David Bühler, and Boris Yakobowski. “Structuring Abstract Interpreters through State and Value Abstractions”. In: *18th International Conference on Verification, Model Checking, and Abstract Interpretation VMCAI 2017*. 2017 (cit. on p. 12).

- [BC11] Richard Bonichon and Pascal Cuoq. “A Mergeable Interval Map”. In: *Stud. Inform. Univ.* 9.1 (2011), pp. 5–37 (cit. on p. 209).
- [Ber+10] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. “Static analysis and verification of aerospace software by abstract interpretation”. In: *Proc. of AIAA Infotech@Aerospace*. AIAA-2010-3385. 2010, p. 38 (cit. on pp. 10, 257, 268).
- [BHTo8] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. “Program Analysis with Dynamic Precision Adjustment”. In: *23rd IEEE/ACM International Conference on Automated Software Engineering ASE*. 2008, pp. 29–38 (cit. on p. 259).
- [BHZo4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. “Widening Operators for Powerset Domains”. In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI*. 2004, pp. 135–148 (cit. on p. 257).
- [Bla+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software”. In: *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*. 2002, pp. 85–108 (cit. on p. 46).
- [Bla+07] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “A Static Analyzer for Large Safety-Critical Software”. In: *CoRR abs/cs/0701193* (2007) (cit. on pp. 33, 46).
- [Bou93] François Bourdoncle. “Efficient chaotic iteration strategies with widenings”. English. In: *Formal Methods in Programming and Their Applications*. Ed. by Dines Bjørner, Manfred Broy, and Igor V. Pottosin. Vol. 735. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1993, pp. 128–141 (cit. on pp. 31, 53).
- [BRo6] Gogul Balakrishnan and Thomas W. Reps. “Recency-Abstraction for Heap-Allocated Storage”. In: *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29–31, 2006, Proceedings*. Ed. by Kwangkeun Yi. Vol. 4134. Lecture Notes in Computer Science. Springer, 2006, pp. 221–239 (cit. on pp. 36, 231).

- [Bra+14] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. “IKOS: A Framework for Static Analysis Based on Abstract Interpretation”. In: *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*. 2014, pp. 271–277 (cit. on p. 46).
- [Bry86] Randal E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691 (cit. on p. 257).
- [C11] ISO Working Group 14. *Programming languages – C. Standard*. ISO/IEC 9899:2011. International Organization for Standardization, 2011 (cit. on pp. 11, 74, 77, 78, 90).
- [C99] *Rationale for International Standard – Programming Languages – C. Standard*. Revision 5.10. International Organization for Standardization, 2003 (cit. on p. 75).
- [CC15] Junjie Chen and Patrick Cousot. “A Binary Decision Tree Abstract Domain Functor”. In: *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*. 2015, pp. 36–53 (cit. on p. 258).
- [CC77a] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. 1977, pp. 238–252 (cit. on p. 26).
- [CC77b] Patrick Cousot and Radhia Cousot. “Static Determination of Dynamic Properties of Generalized Type Unions”. In: *Language Design for Reliable Software*. 1977, pp. 77–94 (cit. on pp. 35, 36).
- [CC79a] P. Cousot and R. Cousot. “Constructive Versions of Tarski’s Fixed Point Theorems”. In: *Pacific Journal of Mathematics* 81.1 (1979), pp. 43–57 (cit. on p. 31).
- [CC79b] Patrick Cousot and Radhia Cousot. “Systematic Design of Program Analysis Frameworks”. In: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*. 1979, pp. 269–282 (cit. on pp. 26, 38, 71, 257, 259, 260).
- [CC92a] Patrick Cousot and Radhia Cousot. “Abstract Interpretation Frameworks”. In: *J. Log. Comput.* 2.4 (1992), pp. 511–547 (cit. on p. 26).

- [CC92b] Patrick Cousot and Radhia Cousot. “Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation”. In: *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP’92, Leuven, Belgium, August 26-28, 1992, Proceedings*. 1992, pp. 269–295 (cit. on pp. 32, 33).
- [CCF13] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. “A Survey on Product Operators in Abstract Interpretation”. In: *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013*. 2013, pp. 325–336 (cit. on pp. 34, 259).
- [CCH00] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. “Combinations of abstract domains for logic programming: open product and generic pattern construction”. In: *Sci. Comput. Program.* 38.1-3 (2000), pp. 27–71 (cit. on p. 42).
- [CCM10] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. “A Scalable Segmented Decision Tree Abstract Domain”. In: *Time for Verification, Essays in Memory of Amir Pnueli*. 2010, pp. 72–95 (cit. on p. 258).
- [CD08] Pascal Cuoq and Damien Doligez. “Hashconsing in an incrementally garbage-collected system: a story of weak pointers and hashconsing in ocaml 3.10.2”. In: *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*. 2008, pp. 13–22 (cit. on p. 210).
- [CH78] Patrick Cousot and Nicolas Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. 1978, pp. 84–96 (cit. on pp. 35, 36).
- [Cil] CIL: Infrastructure for C Program Analysis and Transformation. <https://people.eecs.berkeley.edu/~necula/cil/>. 2009 (cit. on p. 74).
- [Cla+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-Guided Abstraction Refinement”. In: *Computer Aided Verification, 12th International Conference, CAV*. 2000, pp. 154–169 (cit. on p. 259).
- [Coq] The Coq development team. *The Coq proof assistant reference manual*. <http://coq.inria.fr>. LogiCal Project. 2015 (cit. on p. 254).

- [Coro8] Agostino Cortesi. “Widening Operators for Abstract Interpretation”. In: *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*. 2008, pp. 31–40 (cit. on p. 32).
- [Cou+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “The ASTRÉE Analyzer”. In: *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. 2005, pp. 21–30 (cit. on p. 46).
- [Cou+06] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “Combination of Abstractions in the ASTRÉE Static Analyzer”. In: *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*. 2006, pp. 272–300 (cit. on pp. 34, 42, 43, 205).
- [Cou+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. “Why does Astrée scale up?” In: *Formal Methods in System Design* 35:3 (2009), pp. 229–264 (cit. on pp. 33, 46, 230).
- [Cou78] P. Cousot. “Méthodes itératives de construction et d’approximation de points fixes d’opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)”. Thèse d’État ès sciences mathématiques. Grenoble, France: Université Joseph Fourier, 21 March 1978 (cit. on p. 26).
- [Cou81] P. Cousot. “Semantic Foundations of Program Analysis”. In: *Program Flow Analysis: Theory and Applications*. Ed. by S.S. Muchnick and N.D. Jones. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981. Chap. 10, pp. 303–342 (cit. on p. 26).
- [Cou99] Patrick Cousot. “The Calculational Design of a Generic Abstract Interpreter”. In: *Calculational System Design*. NATO ASI Series F. IOS Press, 1999 (cit. on pp. 33, 53).
- [CS12] Loïc Correnson and Julien Signoles. “Combining Analyses for C Program Verification”. In: *Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*. 2012, pp. 108–130 (cit. on p. 52).

- [Cuo+09] Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. “Experience report: OCaml for an industrial-strength static analysis framework”. In: *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. 2009, pp. 281–286 (cit. on p. 53).
- [Cuo+12] Pascal Cuoq, Philippe Hilsenkopf, Florent Kirchner, Sébastien Labbé, Nguyen Thuy, and Boris Yakobowski. “Formal verification of software important to safety using the Frama-C tool suite”. In: *NPIC & HMIT*. 2012 (cit. on p. 69).
- [DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. “ESP: Path-Sensitive Program Verification in Polynomial Time”. In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2002*. 2002, pp. 57–68 (cit. on p. 258).
- [Dup13] Jan Dupal. “Reduced Product of Abstract Domains”. PhD thesis. Masarykova univerzita, Fakulta informatiky, 2013 (cit. on p. 43).
- [Fero4] Jérôme Feret. “Static Analysis of Digital Filters”. In: *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. 2004, pp. 33–48 (cit. on pp. 35, 36).
- [FJM05] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. “Joining dataflow with predicates”. In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. 2005, pp. 227–236 (cit. on pp. 259, 266).
- [FL10] Manuel Fähndrich and Francesco Logozzo. “Static Contract Checking with Abstract Interpretation”. In: *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*. 2010, pp. 10–30 (cit. on p. 46).
- [Gan+13] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. “Abstract Interpretation over Non-lattice Abstract Domains”. In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. 2013, pp. 6–24 (cit. on p. 34).

- [GC10a] Arie Gurfinkel and Sagar Chaki. “Boxes: A Symbolic Abstract Domain of Boxes”. In: *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*. 2010, pp. 287–303 (cit. on pp. [35](#), [36](#), [258](#), [272](#)).
- [GC10b] Arie Gurfinkel and Sagar Chaki. “Combining predicate and numeric abstraction for software model checking”. In: *STTT* 12.6 (2010), pp. 409–427 (cit. on p. [259](#)).
- [GR98] Roberto Giacobazzi and Francesco Ranzato. “Optimal Domains for Disjunctive Abstract Interpretation”. In: *Sci. Comput. Program.* 32.1-3 (1998), pp. 177–210 (cit. on p. [257](#)).
- [Gra89] Philippe Granger. “Static analysis of arithmetical congruences”. In: *International Journal of Computer Mathematics* 30.3-4 (1989), pp. 165–190 (cit. on pp. [35](#), [36](#), [40](#), [130](#)).
- [Gra91] Philippe Granger. “Static Analysis of Linear Congruence Equalities among Variables of a Program”. In: *TAP-SOFT’91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP’91)*. 1991, pp. 169–192 (cit. on pp. [35](#), [36](#)).
- [Gra92] Philippe Granger. “Improving the Results of Static Analyses Programs by Local Decreasing Iteration”. In: *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, Proceedings*. 1992, pp. 68–79 (cit. on p. [40](#)).
- [Gra97] Philippe Granger. “Static Analyses of Congruence Properties on Rational Numbers (Extended Abstract)”. In: *Static Analysis, 4th International Symposium, SAS ’97, Paris, France, September 8-10, 1997, Proceedings*. 1997, pp. 278–292 (cit. on pp. [35](#), [36](#)).
- [GS96] Susanne Graf and Hassen Saïdi. “Verifying Invariants Using theorem Proving”. In: *CAV*. Vol. 1102. LNCS. 1996, pp. 196–207 (cit. on p. [259](#)).
- [Gur+15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. “The SeaHorn Verification Framework”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 2015, pp. 343–361 (cit. on p. [46](#)).
- [Hal93] Nicolas Halbwachs. “Delay Analysis in Synchronous Programs”. In: *Computer Aided Verification, 5th International Conference, CAV ’93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*. 1993, pp. 333–346 (cit. on p. [33](#)).

- [HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. “Software Model Checking for People Who Love Automata”. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 2013, pp. 36–52 (cit. on p. 266).
- [Hin01] Michael Hind. “Pointer analysis: haven’t we solved this problem yet?” In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE’01, Snowbird, Utah, USA, June 18-19, 2001*. Ed. by John Field and Gregor Snelting. ACM, 2001, pp. 54–61 (cit. on p. 36).
- [HMM12] Julien Henry, David Monniaux, and Matthieu Moy. “PAGAI: a path sensitive static analyzer”. In: *CoRR abs/1207.3937* (2012) (cit. on p. 276).
- [HT98] Maria Handjieva and Stanislav Tzolovski. “Refining Static Analyses by Trace-Based Partitioning Using Control Flow”. In: *Static Analysis, 5th International Symposium, SAS. 1998*, pp. 200–214 (cit. on p. 258).
- [Ja] B. Jeannet and al. *The Interproc Analyzer*. URL: <http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html> (cit. on p. 46).
- [JGo8] Patricia Johann and Neil Ghani. “Foundations for structured programming with GADTs”. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 297–308 (cit. on pp. 13, 60).
- [JM09] Bertrand Jeannet and Antoine Miné. “Apron: A Library of Numerical Abstract Domains for Static Analysis”. In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. 2009, pp. 661–667 (cit. on pp. 46, 230).
- [Jou+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. “A Formally-Verified C Static Analyzer”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, pp. 247–259 (cit. on pp. 43, 46, 205).
- [Jou16] Jacques-Henri Jourdan. “Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié)”. PhD thesis. Paris Diderot University, France, 2016 (cit. on pp. 43, 205).

- [Kar76] Michael Karr. “Affine Relationships Among Variables of a Program”. In: *Acta Inf.* 6 (1976), pp. 133–151 (cit. on pp. 35, 36).
- [Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. “Frama-C: A software analysis perspective”. In: *Formal Asp. Comput.* 27.3 (2015), pp. 573–609 (cit. on pp. 12, 46, 51, 69, 209).
- [Knu74] Donald E. Knuth. “Computer Programming as an Art”. In: *Communications of the ACM* 17.12 (1974), pp. 667–673 (cit. on p. iii).
- [KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978 (cit. on pp. 10, 73).
- [LA04] Chris Lattner and Vikram S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 2004, pp. 75–88 (cit. on p. 46).
- [Lar+97] Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. “Efficient verification of real-time systems: compact data structure and state-space reduction”. In: *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*. 1997, pp. 14–24 (cit. on p. 36).
- [Lea+08] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*. 2008 (cit. on pp. 33, 75).
- [Ler09] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (2009), pp. 107–115 (cit. on p. 46).
- [LJG11] Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. “Widening with Thresholds for Programs with Complex Control Graphs”. In: *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*. 2011, pp. 492–502 (cit. on p. 33).
- [Mino01] Antoine Miné. “A New Numerical Abstract Domain Based on Difference-Bound Matrices”. In: *Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings*. Ed. by Olivier Danvy and Andrzej Filinski. Vol. 2053. Lecture Notes in Computer Science. Springer, 2001, pp. 155–172 (cit. on p. 36).

- [Mino6a] Antoine Miné. “Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics”. In: *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’06), Ottawa, Ontario, Canada, June 14-16, 2006*. 2006, pp. 54–63 (cit. on pp. [36](#), [210](#)).
- [Mino6b] Antoine Miné. “Symbolic Methods to Enhance the Precision of Numerical Abstract Domains”. In: *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*. 2006, pp. 348–363 (cit. on pp. [79](#), [216](#), [273](#)).
- [Mino6c] Antoine Miné. “The octagon abstract domain”. In: *Higher-Order and Symbolic Computation* 19.1 (2006), pp. 31–100 (cit. on pp. [35](#), [36](#)).
- [Moo66] Ramon E Moore. *Interval analysis*. Vol. 4. Prentice-Hall Englewood Cliffs, 1966 (cit. on pp. [36](#), [130](#)).
- [MR05] Laurent Mauborgne and Xavier Rival. “Trace Partitioning in Abstract Interpretation Based Static Analyzers”. In: *Programming Languages and Systems, 14th European Symposium on Programming, ESOP*. 2005, pp. 5–20 (cit. on pp. [258](#), [273](#)).
- [MS14] Bogdan Mihaila and Axel Simon. “Synthesizing Predicates from Abstract Domain Losses”. In: *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*. 2014, pp. 328–342 (cit. on pp. [259](#), [266](#)).
- [Nec+02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs”. In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. 2002, pp. 213–228 (cit. on pp. [52](#), [73](#), [74](#)).
- [NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999 (cit. on p. [31](#)).
- [OG98] Chris Okasaki and Andrew Gill. “Fast Mergeable Integer Maps”. In: *In Workshop on ML*. 1998, pp. 77–86 (cit. on p. [228](#)).

- [PCo6] Corneliu Popeea and Wei-Ngan Chin. “Inferring Disjunctive Postconditions”. In: *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference*. 2006, pp. 331–345 (cit. on p. 257).
- [Ric53] Henry Gordon Rice. “Classes of recursively enumerable sets and their decision problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366 (cit. on p. 8).
- [RL12] Valentin Robert and Xavier Leroy. “A Formally-Verified Alias Analysis”. In: *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*. Ed. by Chris Hawblitzel and Dale Miller. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 11–26 (cit. on p. 36).
- [San+o6a] Sriram Sankaranarayanan, Michael Colón, Henny B. Sipma, and Zohar Manna. “Efficient Strongly Relational Polyhedral Analysis”. In: *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*. 2006, pp. 111–125 (cit. on p. 245).
- [San+o6b] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. “Static Analysis in Disjunctive Numerical Domains”. In: *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*. 2006, pp. 3–17 (cit. on pp. 257, 266).
- [SOo8] Matthieu Sozeau and Nicolas Oury. “First-Class Type Classes”. In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. 2008, pp. 278–293 (cit. on p. 255).
- [SRWo2] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. “Parametric shape analysis via 3-valued logic”. In: *ACM Trans. Program. Lang. Syst.* 24.3 (2002), pp. 217–298 (cit. on p. 36).
- [Tar55] Alfred Tarski. “A lattice-theoretical fixpoint theorem and its applications.” In: *Pacific J. Math.* 5.2 (1955), pp. 285–309 (cit. on p. 30).
- [TCR13] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. “Reduced Product Combination of Abstract Domains for Shapes”. In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013,*

- Rome, Italy, January 20-22, 2013. *Proceedings*. 2013, pp. 375–395 (cit. on pp. [36](#), [42](#)).
- [Tur49] Alan Turing. “Checking a large routine”. In: *Report of a Conference on High Speed Automatic Calculating Machines*. University Mathematical Laboratory, Cambridge. 1949, pp. 67–69 (cit. on p. [iv](#)).
- [UM14] Caterina Urban and Antoine Miné. “A Decision Tree Abstract Domain for Proving Conditional Termination”. In: *Static Analysis - 21st International Symposium, SAS*. 2014, pp. 302–318 (cit. on p. [258](#)).
- [VB04] Arnaud Venet and Guillaume P. Brat. “Precise and efficient static array bound checking for large embedded C programs”. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. Ed. by William Pugh and Craig Chambers. ACM, 2004, pp. 231–242 (cit. on p. [36](#)).
- [Ven12] Arnaud Venet. “The Gauge Domain: Scalable Analysis of Linear Inequality Invariants”. In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 139–154 (cit. on p. [231](#)).
- [Yak15] Boris Yakobowski. “Fast Whole-program Verification Using On-the-fly Summarization.” In: *Workshop on Tools for Automatic Program Analysis*. 2015 (cit. on pp. [58](#), [275](#)).
- [Yov96] Sergio Yovine. “Model Checking Timed Automata”. In: *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems, Veldhoven, The Netherlands, November 25-29, 1996*. 1996, pp. 114–152 (cit. on p. [36](#)).