Static analysis of functional programs with an application to the frame problem in deductive verification

Oana Fabiana Andreescu

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

Oana Fabiana Andreescu

préparée à Prove & Run et á l'unité de recherche 6074 – IRISA
Institut de Recherche en Informatique et Systemes Aleatoires

Static Analysis of Functional Programs with an Application to the Frame Problem in Deductive Verification

Thèse soutenue à Rennes
le 29 Mai 2017

devant le jury composé de :

Sandrine Blazy
Professeure / Présidente

Catherine Dubois
Professeure / Rapporteuse

Antoine Miné
Professeur / Rapporteur

Sylvain Conchon
Professeur / Examinateur

Thomas Jensen
Professeur / Directeur de thèse

Stéphane Lescuyer
Ingénieur / Co-directeur de thèse

UNIVERSITÉ DE RENNES 1

Abstract

Prove & Run
École doctorale Matisse

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Static Analysis of Functional Programs with an Application to the Frame Problem in Deductive Verification

by Oana Fabiana Andreescu

In the field of software verification, the frame problem refers to establishing the boundaries within which program elements operate. It has notoriously tedious consequences on the specification of frame properties, which indicate the parts of the program state that an operation is allowed to modify, as well as on their verification, i.e. proving that operations modify only what is specified by their frame properties. In the context of interactive formal verification of complex systems, such as operating systems, much effort is spent addressing these consequences and proving the preservation of the systems' invariants. However, most operations have a localized effect on the system and impact only a limited number of invariants at the same time. In this thesis we address the issue of identifying those invariants that are unaffected by an operation and we present a solution for automatically inferring their preservation. Our solution is meant to ease the proof burden for the programmer. It is based on static analysis and does not require any additional frame annotations. Our strategy consists in combining a dependency analysis and a correlation analysis. We have designed and implemented both static analyses for a strongly-typed, functional language that handles structures, variants and arrays. The dependency analysis computes a conservative approximation of the input fragments on which functional properties and operations depend. The correlation analysis computes a safe approximation of the parts of an input state to a function that are copied to the output state. It summarizes not only what is modified but also how it is modified and to what extent. By employing these two static analyses and by subsequently reasoning based on their combined results, an interactive theorem prover can automate the discharching of proof obligations for unmodified parts of the state. We have applied both of our static analyses to a functional specification of a micro-kernel and the obtained results demonstrate both their precision and their scalability.

Acknowledgements

First of all, I would like to express my gratitude to my two PhD advisors, Thomas Jensen and Stéphane Lescuyer, without whom this thesis would have been impossible. I thank them for their patience and dedication in guiding me throughout these years and for all the rigour that they instilled into me, by word and by their own example. Thomas, thank you for helping me put my work into perspective. Thank you for your encouragement when I was overwhelmed by doubts and for your optimism when I had none. Stéphane, thank you for your inspiring advices, for the rigorous proofreading, for the many interesting discussions and for your careful attention to my work. Know that this thank you note was written using Emacs to which I am happy to admit that you converted me.

I am in debt to Dominique Bolignano for raising the possibility of this thesis and for creating the frame that allowed me to embark on this interesting journey and to explore the seas of research among an inspiring group of professionals - the Prove & Run team.

I am grateful to and would like to wholeheartedly thank Catherine Dubois and Antoine Miné for accepting to review my dissertation. I am honoured to know that my 200+ pages have been read by experts of static analysis and formal verification and I am grateful for their valuable comments and remarks.

I would also like to thank Sandrine Blazy and Sylvain Conchon for accepting to be members of the jury. Sylvain Conchon, I am grateful for your keen interest during my defense. Sandrine Blazy, thank you for accepting to chair my defense and for driving it in such a positive manner!

For their understanding, their advice and their support during the transition period and the months before my defense, I would like to thank Claire Loiseaux and Carolina Lavatelli.

I thank all of my colleagues at Prove & Run for our discussions and their advice during these years. I thank Florence for her warmth, energy and optimism, Erica and Henry for being such great office colleagues, Pauline and François for being friendly, reliable colleagues in the academic trenches. I am in debt to Olivier and Benoit for reviewing my articles and providing valuable remarks. I thank Pascale for smoothing out the stormy waves of administrative work. Though our interactions were briefer, I would like to also thank the Celtique members for their openness and for the interesting seminaries. A special thanks goes to Lydie Mabil for helping me deal with the administrative work during these years, and finally, for helping prepare the defense of my dissertation.

This academic journey started long ago, even before I was aware, with the help of Marius Minea and Ovidiu Badescu, who unknowingly, motivated me to take this path years later. I warmly thank them and I am grateful to both for paving the first part of my academic path.

I would also like to thank my friends, old and new, far and near. Thank you for always being there for me and providing perspective, enthusiasm and breaths of fresh air. Thank you as well for still being my friends despite the long, winded and geeky

descriptions of my work and the occasionally cancelled plans and absences while I was trying to find my way into the research world.

I lack the appropriate words to express the gratitude I feel towards my family for their never-ending love and support. I thank my mother and my sister for being such wonderful examples of women in science. I thank my father for his unwavering belief in me and for his love and respect for well-written sentences, no matter the context, which he instilled into me. I thank my brother-in-law for being the one who ignited early on the sparkle and interest for computers and mathematics, and my two wonderful nieces for always being my rays of light.

Last but surely not least, I have only gratitude for Georges, my companion, my pillar of strength, my compass and lighthouse during the darkest moments. To quote Carl Sagan, in the vastness of space and immensity of time, it is my absolute joy to spend a planet and an epoch with you!

Contents

List of Figures

List of Tables

List of notations

Section	Symbol	Type	Description	
Sec. 3.1.2	true	$\mathcal{L}$	Special exit label	34
Sec. 3.1.2	false	$\mathcal{L}$	Special exit label	35
Sec. 4.1	T_0	$\subset \mathbb{T}$	Set of base type identifiers	60
Def. 4.1.1	$\mathbb{T}$		Universe of type identifiers	60
Def. 4.1.1	τ	$\mathbb{T}$	Type	60
Def. 4.1.1	τ_0	$\mathbb{T}$	Primitive type	60
Def. 4.1.1	$\boldsymbol{struct}\{f_1 : \tau_1, \ldots\}$	$\mathbb{T}$	Structure type	60
Def. 4.1.1	$\boldsymbol{variant}[C_1 : \tau_1 \mid \ldots]$	$\mathbb{T}$	Variant type	60
Def. 4.1.1	$\boldsymbol{arr}^{\tau}\langle\tau\rangle$	$\mathbb{T}$	Array type	60
Sec. 4.1	λ	$\mathcal{L}$	Exit label	61
Sec. 4.1	$\mathcal{L}$		Set of exit labels	61
Sec. 4.1	error	$\mathcal{L}$	Special exit label	61
Sec. 4.1	σ, σ_p	Σ	Signature (of predicate p)	61
Sec. 4.1	Σ		Set of predicate signatures	61
Sec. 4.1	$o, \bar{o}$	$\mathcal{V}$	Output variable(s)	61
Tab. 4.2	s		αSmil statement	62
Tab. 4.2	$o := e$		αSmil assignment statement	62
Tab. 4.2	$e_1 = e_2$		αSmil equality test statement	62
Tab. 4.2	$\boldsymbol{nop}$		αSmil no operation statement	62
Tab. 4.2	$r := \{e_1, \ldots, e_n\}$		αSmil create structure statement	62
Tab. 4.2	$\{o_1, \ldots, o_n\} := r$		αSmil destructure structure	62
Tab. 4.2	$o := r.f_i$		αSmil access field statement	62
Tab. 4.2	$r' := \{r\ \boldsymbol{with}\ f_i = e\}$		αSmil update field statement	62
Tab. 4.2	$r' = \langle f_1, \ldots, f_k\rangle r''$		αSmil partial structure equality	62
Tab. 4.2	$v := C_p[e]$		αSmil create variant statement	62
Tab. 4.2	$\boldsymbol{switch}(v)\ \boldsymbol{as}\ [o_1 \mid \ldots]$		αSmil destructure variant statement	62
Tab. 4.2	$v \in \{C_1, \ldots, C_k\}$		αSmil variant possible statement	62
Tab. 4.2	$o := a[i]$		αSmil array access statement	62
Tab. 4.2	$a' := [a\ \boldsymbol{with}\ i = e]$		αSmil array update statement	62
Tab. 4.2	$p(e_1, \ldots)\ [\lambda_1 : \bar{o}_1 \mid \ldots]$		αSmil predicate call statement	62
Sec. 4.2	$G_p = (N, \mathcal{E})$		Control flow graph of predicate p	67
Def. 4.3.1	Γ	$\mathcal{V} \to \mathbb{T}$	Typing environment	68
Sec. 4.3	v	$\mathcal{V}$	Variable	68
Sec. 4.3	$\mathcal{V}$		Set of variables	68
Sec. 4.3	$\mathcal{V}^+$	$\subseteq \mathcal{V}$	Writable variable identifiers	68
Def. 4.3.2	Σ	$\mathscr{P} \to \mathscr{S}$	Maps predicate ids to signatures	68

To my family and close ones

Chapter I

Résumé étendu en Français

I.1 Le Problème du *Frame*

Dans le domaine de la vérification formelle de logiciels, il est impératif d'identifier les limites au sein desquelles les éléments ou fonctions opèrent. Une spécification complète d'une opération doit non seulement préciser que les valeurs de sortie possèdent une certaine propriète, mais elle doit également délimiter les parties de l'état d'éntrée sur lesquelles l'opération fonctionne. Ces limites constituent les propriétés de *frame* (*frame properties* en anglais). Elles sont habituellement spécifiées manuellement par le programmeur et leur validité doit être vérifiée : il est nécessaire de prouver que les opérations du programme n'outrepassent pas les limites ainsi déclarées. La spécification et la preuve de propriétés de *frame* est une tâche notoiremment connue comme étant longue et fastidieuse. L'effort considérable investi dans cette tâche est une manifestation du problème de *frame* (*frame problem* en anglais). Les manifestations du problème de *frame* apparaissent dans le contexte de tous les langages de spécification et de toutes les méthodes de vérification formelle.

I.2 Objectifs

Au fil du développement de ProvenCore, un micro-noyau polyvalent qui garantit l'isolation, il est apparu évident que la spécification et la vérification des systèmes de transition, en général, ainsi que la spécification et vérification des systèmes d'exploitation en particulier ne sont pas immunes au problème du *frame*. Les systèmes d'exploitation sont caractérisés par des états complexes définis par des types de données algébriques et des tableaux associatifs, qui sont des briques fondamentales pour représenter et manipuler des données complexes d'une manière efficace. Les systèmes d'exploitation sont aussi caractérisés par des transitions, qui associent de tels états d'entrée à de nouveaux états de sortie. Cependant, la plupart des transitions ne sont pas concernées par l'état d'entrée dans son intégralité, mais dépendent de et modifient un sous-ensemble de celui-ci. Intuitivement, des propriétés valides pour l'état d'entrée restent trivialement valides pour l'état de sortie obtenue après la transition, tant qu'elles dépendent seulement des parties de l'état d'entrée qui ne sont pas modifiées par la transition. En pratique, prouver la préservation de ces propriétés n'est pas une tâche évidente, et impose un effort manuel conséquent et une foule de preuves pénibles et répétitives.

L'objectif de notre travail a été d'adresser ce problème et de trouver une solution automatisée pour inférer la préservation de ces propriétés. Plus précisément, notre but a été l'inférence automatique des propriétés qui dépendent d'un sous-ensemble de l'entrée qui est disjoint du *frame* de l'opération, c'est-à-dire du sous-ensemble de l'état qui est modifié. À cette fin, nous avons proposé une solution basée sur l'analyse statique, qui ne requiert pas d'annotations de *frame* supplémentaires. En détectant le sous-ensemble de l'état dont dépend une propriété ainsi que la partie qui n'est pas affectée par une opération, nous pouvons résoudre automatiquement les obligations de preuve liées à des parties non modifiées.

Nous employons deux analyses statiques dans ce but : une analyse de dépendance et une analyse de corrélation. Les deux analyses gèrent des programmes manipulant des tableaux associatifs ainsi que des types de données algébriques (structures et variants), et calculent des résultats reflétant la structure sous-jacente de ces types (champs, constructeurs et cellules de tableau). Un raisonnement automatique basé sur le résultat combiné de ces deux analyses statiques permet d'inférer la préservation de certaines propriétées relatives à l'état de sortie. À terme, ces deux analyses ont pour vocation à être employées par une tactique de preuve qui sera intégrée à l'assistant de preuve interactive inclus dans la suite logicielle ProvenTools, développée par Prove & Run.

Smart, le langage ciblé par la suite logicielle ProvenTools, est un langage purment fonctionnel qui manipule des structures de données algébriques et des tableaux associatifs immuables. Ce travail a été motivé par la vérification de ProvenCore. ProvenCore est implémenté via de multiples raffinements entre des modèles successifs du noyau, du plus abstrait, qui permet la définition et la preuve de la propriété d'isolation, au plus concret, qui est utilisé pour la génération de code. Les états globaux des couches abstraites sont des structures complexes contenant de nombreux champs eux-mêmes composites. Des commandes telles que `fork`, `exec` et `exit` peuvent être exécutées. Chacune de ces commandes reçoit comme argument un état global d'entrée, et produit l'état du système après exécution de la commande. En pratique, la plupart des commandes supportées par le système ne menacent qu'un nombre limité d'invariants. Prouver automatiquement la préservation des invariants immunes peut diminuer considérablement le nombre total de preuves à la charge du programmeur, et permet à celui-ci de se concentrer sur les preuves les plus intéressantes.

I.3 Analyse de dépendance

L'analyse de dépendance gère des fonctions et leur spécification de manière uniforme. Elle calcule conservativement pour chaque scénario d'exécution possible une approximation des sous-éléments de l'état d'entrée desquels dépend le résultat. Pour les variants, une analyse supplémentaire est effectuée simultanément, afin de calculer le sous-ensemble des constructeurs possibles dans chaque scénario d'exécution.

Nous avons défini notre propre domaine abstrait représentant les dépendances, et obtenons des informations de dépendance qui reflètent la structure en couche des types de données.

Cette analyse a été conçue dans le but d'être exécutée à la volée durant la vérification interactive, et opère de manière uniforme sur les programmes et leur spécification, ces deux points conférant à notre approche son originalité. Nous avons implémenté un prototype de cette analyse de dépendance en OCaml et l'avons appliquée à une spécification fonctionnelle de ProvenCore. Les résultats obtenus sont positifs, par exemple l'analyse de dépendance s'exécute en moins d'une seconde sur un ensemble de plus de 600 prédicats totalisant approximativement 10000 lignes de code.

Afin d'introduire pour l'analyse de dépendance une forme de sensibilité au contexte, nous avons conçu une extension basée sur des chemins symboliques. Cette extension rallonge légèrement le temps d'exécution (de 10% à 20% sur les *benchmarks* utilisés). Cependant, en utilisant l'analyse de dépendance avec cette extension, nous avons obtenu des résultats plus précis pour 50% des prédicats inclus dans ces *benchmarks*.

I.4 Anaylse de corrélation

L'analyse de corrélation détecte le flot de valeurs d'entrée dans les valeurs de sortie. Elle calcule conservativement une approximation des équivalences entre les sous-éléments d'entrée et ceux de sortie pour une fonction donnée. C'est une analyse statique interprocédurale qui résume le comportement d'une fonction et qui détecte quelles parties de l'état sont modifiées, et dans quelle mesure. Nous avons défini un type d'équivalence partiel qui reflète la structure des types de données algébriques et tableaux associatifs. Pour gagner en précision, et ne pas perdre d'informations lorsque l'entrée et la sortie ont des types différents, nous avons introduit un niveau intermédiaire. Les corrélations consistent donc en des chemins d'accès vers des sous-éléments de même type, et des équivalences entre ces sous-éléments. Ce niveau intermédiaire permet de calculer de manière flexible des équivalences précises entre des parties de l'entrée et des parties de la sortie.

Nous avons là aussi implémenté en OCaml un prototype de cette analyse de corrélation et nous l'avons appliqué à une spécification fonctionnelle de ProvenCore. Les résultats obtenus sont encourageants : par exemple, les corrélations calculées pour un sous-ensemble de 630 prédicats totalisant approximativement 10000 lignes de code sont obtenus en moins de 0.5 secondes. Bien que plus complexe que l'analyse de dépendance, l'analyse de corrélation s'exécute plus rapidement sur nos *benchmarks* car contrairement à la première, elle ne s'applique qu'aux fonctions, mais pas aux spécifications. En effet, les spécifications sont des prédicats booléens, et ne retournent pas un état modifié.

I.5 Procédure de décision

Nous avons esquissé une procédure de décision qui emploie nos deux analyses statiques. Celle-ci constitue la première étape de notre solution pour l'inférence automatique de la préservation des invariants de *frame*. En mettant au jour des équivalences entre les entrées et les sorties, et après avoir détecté qu'une propriété ne dépend que de

parties inchangées, il est possible d'inférer la préservation des invariants pour ces parties inchangées.

La procédure de décision n'a pas encore été implémentée, mais des expériences préliminaires et un prototype simple nous donnent une idée de la manière dont les résultats de dépendance et de corrélation doivent être unifiés. Par ailleurs, cela nous a permis de déterminer le genre de requêtes qui peuvent être traitées, et le mécanisme permettant d'y répondre. Les résultats obtenus grâce à notre prototype simple sur une spécification fonctionnelle de ProvenCore sont décrits et analysés.

L'unification des résultats des deux analyses passe par la création d'un graphe reliant les variables d'entrée et de sortie examinées par la requête. Les arcs représentent des corrélations entre des sous-éléments de ces variables, qui sont détectées par la seconde analyse. Les dépendances de la propriété dont on cherche à inférer la préservation indiquent les sous-éléments qui influent sur le résultat de cette propriété. Lorsque ces sous-éléments sont laissés intacts, la propriété est trivialement préservée. L'algorithme d'unification parcourt donc le graphe, en tentant de détecter un maximum d'équivalences entre des sous-éléments des variables d'entrée et de sortie. Si les sous-éléments indiqués par la dépendance sont inclus dans l'ensemble des sous-éléments équivalents, alors la propriété est nécessairement préservée, car toutes les valeurs influant sur son résultat sont les mêmes avant et après l'exécution de l'opération.

I.6 Conclusion

Pour conclure, nous avons conçu et implémenté deux analyses statiques qui détectent les dépendances de données d'une propriété logique, ainsi que des corrélations entre les entrées et sorties d'opérations. Nos premiers résultats sur un modèle fonctionnel d'un micro-noyau sont encourageants, tant pour leur précision que pour la vitesse de l'analyse, ce qui rend ces analyses adéquates pour un usage dans le cadre d'un prouveur interactif. Hormis de menues améliorations impactant la précision de notre analyse, les prochaines étapes consistent à les combiner afin de détecter les invariants qui ne sont pas affectés par l'exécution d'un prédicat, puis intégrer cette détection comme tactique dans le prouveur de théorèmes ProvenTools. Nous pensons qu'il est possible de tirer parti des spécifications de *frame* à moindre coût, en particulier sans que cela impose au programmeur l'écriture fastidieuse d'annotations intuitivement évidentes. Lors de la vérification formelle de systèmes de transition complexes, il devient alors possible d'intégrer aux outils de développement une inférence automatique de la préservation des invariants liés au frame, via l'analyse statique.

Chapter 1

Introduction

> No human investigation can claim to be scientific if it doesn't pass the test of mathematical proof.
>
> Leonardo da Vinci

1.1 Formal Verification of Software

Since the middle of the last century, computers and information technology brought forth a digital revolution, fundamentally changing the way we live, work and interact with one another. Nowadays, computer programs govern our world and software permeates our lives in manifold ways, shaping our interactions with the surrounding environment. From the alarm clock that marks the start of our day and the coffee machine that motivates us to leave the house, to the smart phone we use for checking our emails or bank account and the car we are driving (or the automated driverless subway we are relying on), some type of software is discreetly acting in the background. We have grown so accustomed to it that we do not even notice it anymore until it asserts itself by impeding us to check our email, by displaying a blue error screen on an ATM or ticket machine, or by serving us a salty bag of crisps, instead of the desperately needed bottle of water we have just paid for on a vending machine. Such reminders can lead to frustration and cause inconveniences, but essentially they cause minor problems. However, receiving such reminders as a result of malfunctions of medical equipment, such as radiation therapy machines, of flight control systems, Mars orbiters, satellites or nuclear power plants, can have dramatic consequences, endangering human lives, causing environmental harm or entailing significant financial losses. Therefore, the quality of the software around us not only influences the quality of our daily lives, but it might potentially have an impact on our safety and the safety of our surrounding world.

Writing reliable, completely error-free software is a difficult task and even a utopian one in the absence of dedicated rigorous approaches for improving its quality. Indeed, for many software systems no guarantees or warranties are provided and their quality is addressed only by traditional software engineering approaches such as testing or code review which cannot guarantee the absence of bugs. While this can be acceptable for non-critical programs, mission- or safety-critical software systems, for which software

quality is of the utmost importance, have to guarantee the absence of runtime errors and provide high levels of confidence regarding their functional correctness. Certain safety-critical market segments impose standards and regulatory requirements for the development of such software systems. In these domains, formal program verification is emerging as a promising approach, gaining a wider audience and more and more terrain.

Formal program verification comprises a set of techniques and tools that can be used to ensure by mathematical means that the program under scrutiny fulfills its functional correctness requirements, i.e. that it computes the right information. For achieving this goal, a formal description or specification of the program's expected behaviour must be given. Once this is established, multiple mathematical tools can be employed for formally verifying that the program's implementation follows the formal specification.

Formal methods can be traced back to the early days of computer science and their origin can be linked to the names of Floyd (Floyd, 1967), Hoare (Hoare, 1969) and Naur (Naur, 1966), (and later, to that of Dijkstra (Dijkstra, 1976)), and their methods for verifying program code with respect to assertions. Despite their early foundations, formal methods seemed, for decades, to be confined to the research world as a consequence of intricate notations, failure to scale to real-world programs and limited or inadequate tool support. Since the 1960's however, considerable progress has been made in the field of formal methods, in terms of both methodology and tools for computer aided program verification. Still, formal program verification methods are not yet a widespread alternative or even complement to testing in the industry. Unlike testing that cannot show the absence of bugs, the goal of formal verification methods is to prove by means of mathematical tools that the program execution is correct in all specified environments, without actually executing the program itself. These are *static* verification techniques.

Static verification techniques include program typing, model checking, deductive verification methods and static program analysis. Besides requiring a formal specification of the program's intended behaviour and its envisioned properties at runtime, all formal methods are theoretically characterized by *undecidability* and *complexity*, which are addressed by introducing some form of *approximation*. For *soundness* considerations, these approximations are necessarily *over-approximations* and all static verification techniques are necessarily *conservative*: they can prove the absence of some erroneous runtime behaviours but they will inevitably trigger some *false warnings*, rejecting certain behaviours that are in practice correct.

Program Typing. Type systems (Cardelli and Wegner, 1985) are tools for reasoning about programs. More specifically, they constitute "a syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute" (Pierce, 2002). They are used for computing static approximations of the runtime behaviours of the terms in a program and can guarantee that *well-typed* programs are free from certain runtime type errors, such as passing strings as arguments to a primitive arithmetic operation or using an integer as a pointer.

In practice, type systems have become the most widespread instance of formal methods, with applications to many programming languages and automatic *typecheckers* built into a variety of compilers. Static typecheckers entail a variety of benefits ranging from early error detection, to offering convenient abstraction and documentation mechanisms and improving the efficiency of compilers, which nowadays make use of the information provided by typecheckers during their optimization and code generation phases.

The Curry-Howard correspondence implies that types can be used for expressing arbitrary complex mathematical specifications. Additional type annotations could in principle enable the full proof of complex properties, effectively transforming type checkers into proof checkers (Pierce, 2002). Approaches such as Extended Static Checking (Leino, 2001; Leino and Nelson, 1998; Flanagan et al., 2002) made progress towards implementing entirely automatic checks for broad classes of correctness properties.

Additionally, approaches relying on type inference have been used for alias analysis (O'Callahan and Jackson, 1997) and exception analysis (Leroy and Pessaux, 2000). Powerful type systems based on dependent types (Martin-Löf, 1984; Nordström, Petersson, and Smith, 1990) are used in automated theorem proving. Various proof assistants, including Coq (Bertot and Castéran, 2004; Sozeau and team, 1997) [1] are based on type theory.

Model Checking. Model checking is a verification technique exhaustively exploring all possible system states in a systematic manner (Baier and Katoen, 2008). More precisely, given a finite-state model of a system and a formal property, a model checking tool verifies whether the property under scrutiny holds for a state in the given model. Model checking emerged as a popular lightweight formal method as a consequence of progress made in the development of program logic and decision procedures, automatic model checking techniques, and compiler analysis (Jhala and Majumdar, 2009). First, program logic and decision procedures (Nelson and Oppen, 1980; Shostak, 1984) provided the needed framework and algorithmic tools to reason about infinite state spaces. Automatic model checking techniques (Clarke and Emerson, 1981; Vardi and Wolper, 1994) for temporal logic provided algorithmic tools for state-space exploration. Abstract interpretation (Cousot and Cousot, 1977) provided connections between the logical world of infinite state spaces and the algorithmic world of finite representations (Jhala and Majumdar, 2009).

Currently, model checking continues attracting considerable attention from the industry. This can be partly explained by it being a rather general verification approach that is suitable for applications stemming from different areas, ranging from embedded systems to hardware design. In addition, it is also an automatic, lightweight technique supporting partial verification, and requires a low degree of user interaction and a lower degree of expertise (Baier and Katoen, 2008), compared to other verification techniques.

[1] Coq Reference Manual Version 8.6 https://coq.inria.fr/distrib/current/files/Reference-Manual.pdf

Its main weaknesses stem, on one hand, from it suffering from the combinatorial state-space explosion (the number of states needed to model the system accurately may easily exceed the amount of available computer memory), and, on the other hand, from it being less suitable for data-intensive applications.

Model checking techniques also impose the production of models, often expressed using finite-state automata which are in turn described in a dedicated description language. Another prerequisite for model checking is a formal specification of the properties to be verified, typically provided by means of temporal logic, which is suitable for the specification of a variety of properties ranging from functional correctness and safety, to liveness, fairness, and real-time properties (Baier and Katoen, 2008).

Deductive Verification Methods. Deductive verification methods consist in producing formal correctness proofs, by first generating a set of formal mathematical proof obligations from the program and its specification, and by subsequently discharging these. Based on the manner in which proof obligations are discharged, namely automatically or interactively, the deductive verification methods can be classified into two broad categories. Both require a thorough understanding of the system to be proven, as well as a good knowledge of the employed proof tools.

The first category of deductive methods rely on standalone tools, that accept as inputs, programs written in a specific programming language (such as Java, C or Ada) and specified in a dedicated annotation language (such as JML or ACSL). These automatically produce a set of mathematical formulas, called verification conditions, which are typically proven using automatic theorem provers (Gallier, 1987) or satisfiability modulo theories solvers (SMT), such as Alt-Ergo, Z3, CVC3, Yices. Deductive verification tools such as Why3 or Boogie, have their own programming and specification language (WhyML and Boogie, respectively), which can act as intermediate verification languages and are designed as a layer on which to build program verifiers for other languages. Verifiers for C, Dafny, Chalice and Spec# have been built using Boogie. WhyML has been used for the verification of Java, C and Ada programs.

The second category of deductive methods relies on interactive theorem provers (Bertot and Castéran, 2004), also called proof assistants, such as Isabelle, Coq, Agda, HOL or Mizar. Both the program and its specification are encoded in the proof assistant's own language (Gallina and Isar, respectively), and the proofs that a program follows its specification, i.e. that it is functionally correct, are typically conducted in an interactive manner, using the underlying proof construction engine. In other words, users are required to actively participate in the verification process, by providing inductive arguments and guiding the proof through proof tactics, proof hints or strategies.

Both deductive verification methods offer a high level of assurance. For automatic theorem provers, the proof chain consisting of multiple steps (the model of the input programming language, the generator of verification condition, the used SMT solver) at which errors could potentially infiltrate, can be perceived as a weakness. For interactive theorem provers, the high-level expertise required to employ them can be perceived as discouraging by the wider audience. However, major industrial breakthroughs have been recently achieved. For instance, Hyper-V, Microsoft's hypervisor for highly secure

virtualization was verified using VCC and the Z3 prover (Leinenbach and Santen, 2009). CompCert (Leroy, 2009), the first formally proven C compiler, was verified using the Coq proof assistant. High security properties of the seL4 microkernel (Klein et al., 2009) have been proven using the Isabelle/HOL proof assistant.

Static Program Analysis. Static program analysis comprises multiple techniques for computing, at compile-time, safe approximations of the set of values or behaviours that can occur dynamically when executing a program. Static analysis techniques initially emerged in the field of compilation, where they provided manners to generate code efficiently, by avoiding redundant or superfluous computations (Nielson, Nielson, and Hankin, 1999).

Static analyses compute sound, conservative information. However, for decades, their scalability to industrial-size programs has been doubted, and their application has been considered as being limited to the research world and to small programs. Recent major breakthroughs have been achieved however, and they triggered, on one hand, the inclusion of static analysis at different levels of the software validation process (Cousot, 2001) and, on the other hand, a proliferation of static code analysers for a variety of languages, targeting mainstream usage and offering a solution for detecting and eliminating common runtime errors. A recent example is Infer (Calcagno and Distefano, 2011), an open-source static analysis tool for bug detection in Java, C, and Objective-C code. It was developed at Facebook, where it is used as part of the development process for mobile applications. Furthermore, static analysis techniques and tools are nowadays employed in the safety-critical market segment. For instance, Astrée (Cousot et al., 2005; Blanchet et al., 2003; Cousot et al., 2007), a static analyser for embedded software written in C, has been employed for the verification of aerospace software (Delmas and Souyris, 2007; Bouissou et al., 2009; Bertrane et al., 2015). In particular, it has been used for proving the absence of runtime errors in the primary flight control software of the fly-by-wire system of Airbus airplanes.

It is argued (Cousot and Cousot, 2010) that model checking, deductive verification and static program analysis represent approximations of the program semantics formalized by the abstract interpretation theory (Cousot and Cousot, 1977).

Broadly speaking, this thesis focuses on *static program analysis* techniques that are meant to be used during *interactive theorem proving*, in order to facilitate and automate the verification of a certain class of properties, in the context of a *strongly typed* language.

1.2 The Frame Problem in a Nutshell

The frame problem (McCarthy and Hayes, 1969) has been initially identified and described by McCarthy and Hayes in 1969 in the context of Artificial Intelligence (AI). Its history is essentially intertwined with that of logicist AI, the branch of AI attempting

to formalize reasoning within mathematical logic. The initial description of the frame problem is the following:

> "In proving that one person could get into conversation with another, we were obliged to add the hypothesis that if a person has a telephone he still has it after looking up a number in the telephone book. If we had a number of actions to be performed in sequence we would have quite a number of conditions to write down that certain actions do not change the values of certain fluents. In fact, with n actions and m fluents we might have to write down mn such conditions."

Unsurprisingly, given its identification in the context of logicist AI, the frame problem manifests itself in the realm of formal software specification and verification as well (Borgida, Mylopoulos, and Reiter, 1993). In this area, it continues to identify a current problem having notoriously tedious consequences and imposing a considerable amount of manual effort. For instance, when considering a simple procedure:

`transferAmount(ownerId, id1, id2, amount)`

that records the transfer of a given sum of money `amount` from a customer's (identified by `ownerId`) current deposit account (identified by the account number `id1`) to a savings account (identified by the account number `id2`), a reasonable specification would be the following:

Precondition : `owner(id1) = ownerId` $\wedge$ `owner(id2) = ownerId`
$\wedge$
`availableAmount(id1)` $\geq$ `amount`

Postcondition : `availableAmount(id1)'` $=$ `availableAmount(id1) - amount`
$\wedge$
`availableAmount(id2)'` $=$ `availableAmount(id2) + amount`

The program states prior to the procedure's execution and the ones subsequent to it, are referred to by the typical unprimed/prime notation and by the `availableAmount(id)` and `owner(id)` functions. The given specification declares a precondition that has to hold prior to transferring the indicated sum of money from one account to the other and it stipulates that the customer identified by `ownerId` must be the owner of both accounts involved in the transaction. It also requires that the currently available amout of money in the deposit account identified by `id1` is higher than the amount to be transferred. The postcondition specifies the procedure's effects on the final program state and encompasses the conditions that have to hold after executing the procedure. They include a stipulation about incrementing the amount of money available in the savings account by the transferred sum `amount`, as well as one referring to decrementing the amount of money available in the current account by the same amount.

As discussed by Borgida et al. (Borgida, Mylopoulos, and Reiter, 1993), the principles on which this specification relies are simple and ubiquitous. Program states

are represented in terms of predicates and functions, and a procedure's effects on the program state are represented as changes to one or more of these predicates and functions. However, the above specification can be interpreted in at least two manners and multiple implementations, with different effects can comply to it. For instance, one implementation that can be considered results in exactly two changes to the program state, as required by the postcondition and as intuitively expected. Another implementation considered makes these two changes, but additionally also changes the ownership of the two accounts involved in the transition. The postcondition still holds after executing the second procedure version. However, the intuitive interpretation of the given specification, namely that *nothing else* but the amount of money in the two accounts changes, is inconsistent with the second implementation which does more than it is necessary and indeed, even desired. In order to prevent such situations, the postcondition for the `transferAmount(ownerId, id1, id2, amount)` procedure would have to also include conditions such as:

$$\begin{array}{l}\texttt{forall id.}\ \ \texttt{owner(id)'} = \texttt{owner(id)} \wedge \texttt{owner(id2)'} = \texttt{owner(id2)} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\ \texttt{forall id.}\ \ \texttt{id} \mathrel{!}= \texttt{id1} \Rightarrow \texttt{id} \mathrel{!}= \texttt{id2} \Rightarrow \texttt{amount(id)'} = \texttt{amount(id)}\end{array}$$

In other words, the postcondition should include not only information about what changes, but also about what does not change. While this might not seem dramatic for the trivial example illustrated above, in real-world examples this quickly escalates, leading to the necessity of specifying a plethora of conditions of the same type as the ones indicated above. These are called *frame properties*. Writing such conditions is necessary but also notoriously repetitive and tedious. Kogtenkov et al. (Kogtenkov, Meyer, and Velder, 2015) rightfully state that:

> "It is hard enough to convince programmers to state what their program does; forcing them in addition to specify all that it does not do may be a tough sell."

The tedious, undeserved, manual effort entailed by the specification and verification of frame properties is a manifestation of the frame problem. Though certain conventions and approaches, such as the *implicit frames* approach, for specifying frame properties can alleviate the manual effort imposed, some manifestation of the frame problem will be visible to some extent in the context of any specification language and verification method.

1.3 Prove & Run: Objectives and Products

The proliferation of mobile devices, with unprecedented processing power, storage capacity, and access to information, already generated a plethora of new possibilities for billions of people. Breakthroughs in emerging technology stemming from fields such as artificial intelligence and the Internet of Things have increased the number of such

possibilities, but also brought forth an unprecedented number of massive security risks and challenges. Prove & Run's[2] objective is to offer solutions for the security challenges entailed by the large-scale deployment of mobile and connected devices and of the Internet of Things.

Attempts at addressing security challenges and diminishing or eliminating potential security issues in systems linked to such devices must put their underlying operating systems and kernels at the core of their efforts to ensure the absence of errors or faulty behaviours. Any software running on the operating system depends on the operating system. Furthermore, operating systems run in privileged modes, in which protection from certain faulty behaviours is non-existing and bugs can lead to arbitrary effects. Therefore, these central software parts need to provide a high level of trust and demonstrate proven and auditable compliance with security properties.

Motivated by the desire to integrate the usage of formal methods in the industry world and therefore to contribute to the increase of software quality and security, the company's initial efforts concentrated on offering a reliable software solution that facilitates the formalization of software functioning and mathematically proves that this software accurately and correctly follows its specification and ensures complex security properties. This led to the development of ProvenTools, a software development toolchain, designed to write and formally prove models written in Smart, Prove & Run's purely functional, unified programming and specification language. For formally proving models written in Smart, ProvenTools integrates an interactive proof assistant, which automates simple proofs and guides or assists users during more complex ones. The prover was designed to offer detailed explanations about its results, providing either the reasoning steps employed for achieved proofs or detailed information for properties that cannot be proven. Such transparency on the prover's side is imperative for products that have to be certified, as auditors need to be able to verify the claims of the prover. Furthermore, ProvenTools includes a generator for transforming programs modeled in Smart into their equivalents in other languages, such as C, while leveraging the proof guarantees of the Smart model.

Following the development of ProvenTools, Prove & Run reached a new stage, concentrating on developing and providing formally proven microkernels and hypervisors. Unlike the widely used operating systems which are enormous and typically have millions of lines of code, microkernels are compact, minimal software systems, that can provide all the mechanisms that need to run in privileged mode, including low-level address space management, thread management and inter-process communication. They can be used for creating a protected, secure environment on the execution platform, on top of which sensitive, security-critical services can run. Being much smaller in size compared to traditional operating systems, they are amenable to formal verification. Hypervisors or virtualization platforms create and host virtual machines. They create the possibility of running multiple, different operating systems, whose execution is managed by the hypervisor, which has full control over all critical resources, such as the memory or the CPU. Therefore, any security issue of the hypervisor impacts every

[2] Prove & Run Website http://www.provenrun.com/

operating system it hosts. The security and reliability of the host hypervisor is thus crucial.

By employing Smart and ProvenTools, two microkernels have been developed[3]. The first, named *ProvenCore*, is a formally proven general purpose microkernel that ensures isolation, i.e. integrity and confidentiality. The second, named *ProvenCore-M*, targets embedded devices based on microcontrollers. *ProvenVisor* is a hypervisor currently in development at Prove & Run.

1.4 Context and Problem Statement

During the development of ProvenCore it became obvious that the specification and verification of transition systems in general, and operating systems in particular, are not insulated from the frame problem. The latter are characterized by complex states defined by algebraic data types and associative arrays, which are fundamental building blocks for representing, grouping and handling complex data efficiently. Transitions, their other characteristic component, map such a complex input state to an output state. However, most transitions are rarely concerned with the entire input state that they are manipulating for retrieving the output state. Most frequently, they depend on

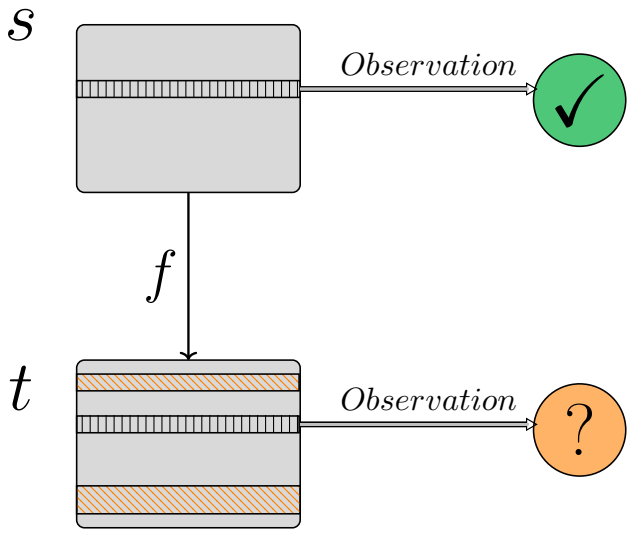


FIGURE 1.1 – Complex Transition Systems: Frame Problem

and modify only a limited subset of it. Intuitively, properties holding for the input state, should hold for the output state following the transition as well, as long as they depend only on fragments of the state that are not modified by the transition. In practice, proving the preservation of such properties does not come for free and imposes considerable manual effort and a multitude of tedious, repetitive proofs.

[3]Prove & Run Products http://www.provenrun.com/products/

This general case is illustrated in Figure 1.1, where a transition system and a state s in it are considered. For the state s, a property depending only on a limited subset, shown in the grey rectangle with vertical lines, is known to hold. A transition f leads to a new state t, obtained by modifying only a small part of the input state s, shown in the orange rectangles with inclined lines. Since the previously proven property is known to depend only on an unmodified subset of the state, we should be able to infer the preservation of the property for the state t as well. This however is not inferred by default.

The goal of this work is to address this issue and to find an automatic solution for inferring the preservation of such properties. More specifically, we target the automatic inference of properties that depend only on an input subset that is disjoint from an operation's *frame*, i.e. the state subset it modifies.

To this end, we propose a solution based on static analysis which does not require any additional frame annotations. We argue that by detecting the subset on which a property depends and by uncovering the part that is not modified by an operation, as shown in Figure 1.2, we can automatically discharge proof obligations related to unmodified parts. We employ two different static analyses for this goal.

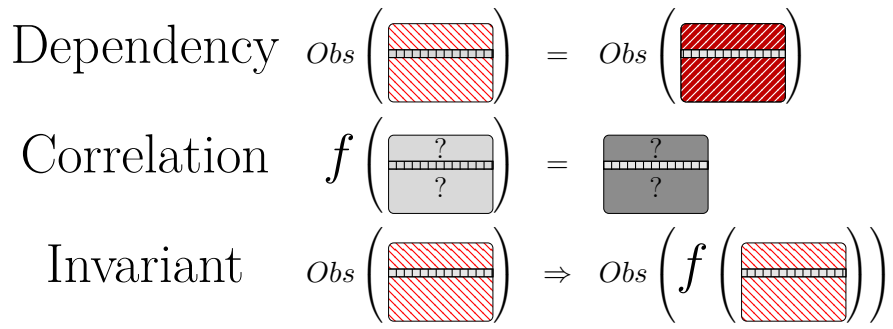


FIGURE 1.2 – Frame Problem and Solution Strategy

The first analysis of our two-step strategy is a *dependency analysis*, which is meant to detect the input subset δ on which the outcome of an operation or of a logical property $\mathscr{L}$ relies. This was illustrated by the grey rectangle with vertical lines in Figure 1.1. The second one, is a *correlation analysis*, meant to detect the subset ξ modified by an operation $\mathscr{O}$. This was illustrated by the orange rectangles with inclined lines in Figure 1.1. By employing these two static analyses, thus detecting δ and ξ automatically, and by subsequently reasoning based on their combined results, we can infer the preservation of the property $\mathscr{L}$ for the post-state of $\mathscr{O}$.

We target the development of a proof tactic that relies on our solution based on static analysis and that is meant to be integrated into the interactive proof assistant offered by ProvenTools. Smart, the language to which the ProvenTools toolchain is associated, is a purely functional language, manipulating immutable algebraic data structures and associative arrays.

The motivation and ideas behind this work were triggered by the verification of *ProvenCore*. Its proof is based on multiple refinements between successive models, from the most abstract, on which the isolation property is defined and proven, to the most concrete, i.e. the actual model used for code generation. The global states of the abstract layers are complex structures with multiple compound fields. Commands such as `fork`, `exec`, `exit` can be executed. Each of these receives as input the global state before executing the command and returns the state of the system after execution. In practice, most supported commands effectively affect only a limited number of invariants. Automatically proving the preservation of unaffected invariants can diminish the total number of proof obligations.

1.5 Contributions and Structure of the Document

We propose an approach for automatically inferring the preservation of framing-related invariants, which is meant to be used in the context of an interactive theorem prover. Our approach employs two different static analyses, namely a dependency analysis and a correlation analysis. Both analyses handle associative arrays and algebraic data types, i.e. structures and variants, and compute fine-grained results mirroring the layered structures of such types.

The dependency analysis handles functions and their specifications in a unified manner and computes for each possible execution scenario a conservative approximation of the input (sub)elements on which their outcome depends. It is a flow-sensitive, path-sensitive interprocedural analysis. For variants, an additional analysis is simultaneously conducted for computing the subset of possible constructors on a given execution scenario.

In order to introduce a relaxed form of context-sensitivity for our dependency analysis, we have devised an extension based on symbolic paths.

The correlation analysis detects the flow of input values into output values. It computes a conservative approximation of fine-grained equivalences between the input and the output subelements of a function. It is an interprocedural analysis that summarises the behaviour of functions and detects what is modified and to what extent.

For both analyses a prototype has been implemented and applied to a medium-sized functional specification of a microkernel.

The rest of this dissertation is structured into 8 chapters, the first two being introductory.

Chapter 2 discusses the manifestations and effects of the frame problem on both formal specification and formal verification and presents some of the main approaches employed for addressing them. We also include a brief presentation of some of the leading specification languages and deductive verification tools and their mechanisms for dealing with frame properties.

In Chapter 3, we introduce the features and the syntax of Smart, the unified programming and specification language developed at Prove & Run and give a concise overview of ProvenTools, the toolchain associated with it.

After these two preliminary chapters, in Chapter 4 we focus on the computational version of Smart's intermediate language, as it is the language that we consider throughout the rest of this dissertation. We present its syntax, underline its specificities and present its formal semantics.

Chapter 5 is dedicated to the dependency analysis, the first of the two static analyses that we have developed and designed as companion tools to be used during interactive program verification. We present our abstract dependency domain that mirrors the layered structure of associative arrays and algebraic data types, discuss the analysis at an intra- and interprocedural level and present the semantic interpretations of the computed dependency information.

Chapter 6 touches upon the issue of context-sensitivity and presents our extension to the dependency analysis presented in Chapter 5. This is meant to eliminate some imprecision by introducing a relaxed form of context-sensitivity.

The correlation analysis, the second component of our strategy for inferring the preservation of frame-related invariants, is presented in Chapter 7. We introduce our abstract partial equivalence type, discuss the need for an additional level of abstraction, allowing us to refer not only to variables, but also to substructures within them, and give an in-depth presentation of the analysis at an intraprocedural level, and a description of it at the interprocedural level.

The implementations of our two analyses and the results obtained on a medium-sized functional specification of a microkernel are presented in Chapter 8. The strategy for employing the information computed by the two analyses is discussed and illustrated.

Finally, Chapter 9 concludes this dissertation with a summary of our contributions and some remarks concerning the specificities of each of our static analyses, as well as our experience with their design and implementation. In addition, we also discuss future perspectives and potential extensions to this work.

Notes about Chapter 5 and Chapter 7

- The work presented in Chapter 5 was the subject of a publication in the proceedings of the 17th International Conference on Formal Engineering Methods (ICFEM15) (Andreescu, Jensen, and Lescuyer, 2015).
- The work presented in Chapter 7 was the subject of a publication in the proceedings of the 14th International Conference on Software Engineering and Formal Methods (SEFM) (Andreescu, Jensen, and Lescuyer, 2016).
- **On-line dedicated web pages.** The prototypes for each of the two discussed static analyses can be tested on their dedicated web pages. Various examples are provided and explained and additionally, users can devise and test their own examples. The corresponding links are indicated in the chapters.

Chapter 2

The Frame Problem in Software Verification

> All his successors gone before him have done't; and all his ancestors that come after him may.
>
> William Shakespeare

In this chapter, in Section 2.1 we give a very brief, necessarily incomplete, presentation of some of the major existing specification languages and verification tools, focusing on those which have addressed the frame problem explicitly and which are relevant for our discussion in the section following it. We then discuss the manifestations of the frame problem in formal specification and verification in Section 2.2 and present the basic approaches to specifying and verifying frame properties in Section 2.3. In Section 2.4 we explain some of the difficulties entailed by these goals, when combined with other concerns such as considerations regarding heap modifications and information hiding. Even though we are not concerned with information hiding and heap modifications are beyond the scope of our work, there are some parallels that can be drawn and some ideas stemming from work that has been done in these areas that are relevant for our context and solution as well. In Section 2.5 we briefly present other approaches to the automatic detection of frame properties. Finally, we give a short overview of some of the approaches used for specifying and reasoning about pure methods in Section 2.6.

2.1 Specification Languages and Verification Tools

Dafny. Dafny (Leino, 2010) is a programming language designed at Microsoft with a focus on verification. It is an imperative, sequential language, supporting generic classes, dynamic allocation and inductive data types. Additionally, it also offers built-in specification constructs, such as pre- and postconditions, frame specifications (which we will discuss in more detail in Section 2.3), quantifiers, loop invariants, and termination metrics (**decreases** clauses used in conjunction with loop invariants). These are reminiscent of contracts in Eiffel (Meyer, 1997; Meyer, 1991), or similar constructs in JML (Leavens, Baker, and Ruby, 2006) and Spec# (Barnett et al., 2005b), which we will present in the following paragraphs as well. Additionally, Dafny also includes

support for algebraic data types, recursive functions and types, as well as updatable ghost variables, which are not allowed to flow into non-ghost variables. Ghost variables and specification constructs in general are eliminated from the executable code, as they are meant to be used strictly during verification. For framing, Dafny relies on dynamic frames (Kassios, 2006) using ghost variables. We will discuss this approach in Section 2.4.

Dafny has an accompanying static program verifier, run as part of the compiler, which targets the verification of functional correctness properties of programs. This is built on top of the Boogie verification engine (Barnett et al., 2005a), which in turn uses Z3 (Moura and Bjørner, 2008). The Dafny compiler translates verified programs written in Dafny to executable code for the .Net Platform. The tool is open source and can be tried online [1].

Smart, the modeling language developed at Prove & Run will be presented in detail in Chapter 3. Similar to Dafny, it is a unified programming and specification language designed with the goal of facilitating verification. Unlike Dafny, Smart is a functional language, relying on *predicates*, the equivalent of functions in other programming languages. Both Dafny and Smart are translated into intermediate languages (Boogie and Smil, respectively), which act as median layers between Dafny or Smart programs and the underlying verification tools. For Smart, the deductive verification tool is an interactive proof assistant. Executable code can be generated from both verified Dafny and verified Smart models.

Spec#. The Spec# programming system (Mike Barnett, 2005; Barnett et al., 2005b; Barnett et al., 2011) includes a programming language, a compiler and a static program verifier. It stems from a research effort focusing on the development of a specification methodology for object-oriented languages and seeking suitable approaches for enforcing it both statically and dynamically. The Spec# methodology introduced some new ideas that influenced the research community and served as a starting point for other approaches (Barnett et al., 2011). It supports sound modular verification of object invariants in the presence of multi-object invariants, subclassing and reentrancy. Spec# led to advances concerning the specification of pure methods, i.e. methods without side-effects, and it introduced an ownership model that allows expressing and using heap topologies in specifications (Barnett et al., 2011). We will discuss the latter in Section 2.4.

The language Spec# is a formal object-oriented language extending the type system of C# with non-null types and checked exceptions. It provides standard method contracts based on pre- and postconditions, as well as object invariants, as inspired by Eiffel and the *Design by Contract* (Meyer, 1992) approach. The accompanying compiler performs various static data-flow analyses for checking that the non-null type system is enforced and that contracts are pure, i.e. have no side-effects. In addition, it also performs *admissibility checks* which are important for soundness and consist in

[1] Dafny Web Page: https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/
Accessed: 2017-02-12. (Archived by WebCite® at http://www.webcitation.org/6oE9sn0iL)

restricting what can appear in object invariants and what pure methods can read. The compiler also emits runtime checks: run-time assertions are generated for the program points at which contracts are supposed to hold and any failure causes an exception to be thrown (Barnett et al., 2011).

Another important contribution having its origins in the Spec# project, are the Boogie intermediate language and verification engine. Spec# programs are translated to the Boogie language, where the heap is modeled as a two-dimensional array indexed by object references and field names. Method calls are modeled by assuming their preconditions and type information, by assigning arbitrary values to anything that they might modify and by subsequently assuming their postconditions. Based on this, verification conditions are generated and expressed in a standard format supported by automatic theorem provers. Any error reported by the theorem prover is mapped back to Boogie and then to Spec# (Barnett et al., 2011).

Spec#[2] has been developed at Microsoft and is publicly available.

Boogie. The Boogie project [3] comprises both an intermediate verification language and a verification tool. The Boogie language (*This is Boogie 2, Boogie Reference Manual*) is meant to be used as an intermediate representation for static program verifiers of various source languages such as Dafny, Chalice, and Spec#. Verifiers for C, such as VCC and HAVOC, have been built on top of Boogie as well. It supports mathematical (types, constants, functions, axioms) and imperative components (global variables, procedure declarations and implementations). The latter specify sets of execution traces, thereby describing and constraining states using the former. Parametric polymorphism, partial orders, nondeterminism, logical quantifications, total expressions and partial statements are among the language's features.

The Boogie verification tool (Barnett et al., 2005a) infers invariants of the input Boogie programs and then generates verification conditions expressed as formulae in first-order logic and arithmetic that are passed to an SMT solver such as Z3. The encoding for the verification formulae allows the reconstruction of error traces from failed proofs.

JML. The *Java Modeling Language* (JML) (Leavens, Baker, and Ruby, 2006; Leavens et al., 2006) is a behavioural interface specification language (Wing, 1987) targeting, as its name implies, the specification of Java classes and interfaces. Its design was guided by the syntax and semantics of Java, as some of the main targeted characteristics were understandability and a shallow learning curve for programmers already familiar with Java. The constructs it supports are inspired by the Design by Contract approach, as well as by the Larch family of specification languages (Guttag, Horning,

[2]Spec# Web Page: https://www.microsoft.com/en-us/research/project/spec/
Accessed: 2017-02-12. (Archived by WebCite® at http://www.webcitation.org/6oEAJnY8b)

[3]Boogie Web Page: https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/
Accessed: 2017-02-12. (Archived by WebCite® at http://www.webcitation.org/6oEAgwOzp)

and Wing, 1985). It also includes quantifiers, constructs for specifying frame conditions and specification-only fields and methods.

Nowadays, an evergrowing variety of tools supports JML (Burdy et al., 2005), ranging from tools for type-checking specifications (the `jmlc` compiler), to tools for runtime debugging, static analysis (such as ESC/Java2 (Flanagan et al., 2002; Burdy et al., 2005; Chalin et al., 2005) and Chase) and verification (such as LOOP, KeY and KRAKATOA).

ESC/Java2 performs extended static checking (Flanagan et al., 2002) for Java programs annotated with specifications written in JML. It can check assertions and detect frequent types of errors in Java, such as dereferencing null or indexing an array outside its bounds. However, the ESC/Java2 tool did not initially address aspects related to checking frame conditions and this became a notorious source of unsoundness (Burdy et al., 2005). Various static verification tools (Berg and Jacobs, 2001; Cataño and Huisman, 2003; Marché, Paulin-Mohring, and Urbain, 2004; Marché, 2016) and dynamic approaches (Lehner and Müller, 2010) addressed this issue.

2.2 Manifestations of the Frame Problem

In the realm of software verification, the frame problem refers to establishing the boundaries within which program elements operate and it has notoriously tedious implications and consequences along two different axes: the specification of *frame properties* or *frame conditions*, which indicate which parts of the program state an operation is allowed to modify, and their verification, i.e. proving that operations modify only what is allowed according to the specified frame properties. Additionally, the verification of frame properties has other ramifications, such as proving the preservation of properties concerning parts of the state that are external to an operation's frame, i.e the parts of the state modified by the operation. Though identified decades ago, in 1969 in the context of Artificial Intelligence (McCarthy and Hayes, 1969), the frame problem is still a current concern in the field of formal specification and verification. Leavens et al. (Leavens, Leino, and Müller, 2007) identify it as one of the difficult remaining challenges in program verification. Even more recently, Bertrand Meyer described it as a subsisting problem (Meyer, 2015). He argues that it constitutes an excellent candidate for automation and describes the usual approaches to the frame problem, such as those frequently based on separation logic (Reynolds, 2005) or ownership types (Clarke, Potter, and Noble, 1998), as elegant, but requiring undeserved manual specification effort, in addition to annotations on the implementation side. In order to make verification appealing to a wider audience in the industry, the amount of annotations required from the programmers is of the utmost importance and thus, must be carefully taken into consideration when devising a solution. While it is legitimate to require the specification of properties expressing the functional behaviour expected of program elements, intermediate properties to which frame properties belong to, should as much as possible be detected automatically. They are an integral

part of a complete specification and they are necessary for proving functional correctness, but in practical terms, they are repetitive and cumbersome and their specification is an inconvenience (Meyer, 2015). Borgida et al. provide a comprehensive discussion of the problem itself and the approaches to addressing it (Borgida, Mylopoulos, and Reiter, 1993; Borgida, Mylopoulos, and Reiter, 1995). In (Borgida, Mylopoulos, and Reiter, 1995), Borgida et al. suggest grouping the permissions to modify variables around variables themselves instead of methods. However, this type of specifications have an unclear semantics in terms of proof obligations (Müller, 2002). A more recent discussion of framing is provided by Hatcliff et al. and it is included in a comprehensive survey of behavioural interface specification languages (Hatcliff et al., 2012). A discussion regarding the remaining challenges related to the frame problem, with a focus on modular verification and information hiding, is included in (Leavens, Leino, and Müller, 2007). The authors discuss possible approaches for addressing these challenges, as well as their respective limitations. In the following section we present the main existing approaches to specifying frame properties.

We remark that Smart does not provide any explicit specification constructs for frame conditions. It is a functional language and it does not support global variables or destructive updates. Implicitly, Smart predicates may read anything passed to them as an input, without modifying it, and write everything in their output or locally declared variables. The preservation of a frame property, i.e. a logical property depending only on parts of the input that are copied without any modification to the output, can be specified as an implication of the form:

frame_property(input) $\implies$ predicate(input, output) $\implies$ frame_property(output)

which can be included either in the predicate's postcondition or as a separate predicate with a Boolean result, receiving the predicate's input, output elements as inputs.

2.3 Approaches to Specifying Frame Properties

Various approaches for expressing frame properties have emerged. These are known as the *manual*, *exclusive*, and *implicit* approaches (Meyer, 2015). We remark that all three major approaches target only the specification of *write effects* of an operation. Most specification languages do not offer special constructs for the specification of *read effects* (some notable exceptions are JML, Dafny and WhyML, the programming and specification language provided by Why3).

2.3.1 The Manual Approach

One of the existing approaches to specifying frame properties does not rely on any specific technique, but instead treats them like any other specification component. This consists in explicitly stating for each operation what is not modified, implicitly conveying that everything else may change. This type of specification can be done with logical variables or with ***old*** expressions by explicitly stating for each unchanged

variable that its value in the operation's post-state is equal to its prior value in the operation's pre-state.

As described by McCarthy and Hayes (McCarthy and Hayes, 1969), with m operations such as *transfer* and n "fluents" such as *owner* in our introductory example from Section 1.2, the manual convention leads to a proliferation of clauses that need to be specified. Their number can potentially be as high as mn. This can prove to be tedious, repetitive and diverting attention and effort from what is truly interesting: what is actually modified by the operation and how. Moreover, this approach can lead to instability in the software process (Meyer, 2015).

For instance, adding new fields to a class whose existing methods are not affected by the newly added fields, requires modifying the postcondition for each existing method and adding clauses of the form *newField* = ***old*** *newField* for each added field.

Both Dafny (Leino, 2010) and Spec# (Leino and Müller, 2008a) support clauses of the form e = ***old****(e)* in method postconditions, for specifying that a method has no impact on the value of an expression e. However, these are not the primary mechanisms for specifying frames in either Dafny or Spec#, as we will discuss in Section 2.3.2.

In Smart, for predicates manipulating inputs and outputs of the same structured type it can be specified in the postcondition that the values of certain fields are equal between the received input and the obtained output. For instance, for a predicate receiving an input structure of type stype, having fields f, g, h and returning an output structure of the same type, where the values of the fields f, h are equal to their values in the input, a standard postcondition would have the following form:

```
stype@equals[f,h](input, output)
```

This can be viewed as a form of old expressions. However, the construct used in the above postcondition, which we will discuss in Chapter 3, was not introduced specifically for this purpose. This idiom is frequently employed for specifying contracts for *implicit predicates*, a form of foreign or native functions signatures.

As we will discuss in Chapter 7, the fine-grained relations that we are detecting between parts of the input and parts of the output can be seen as clauses of the form *subvalue* = ***old***(*subvalue*). However, in our case, these are detected automatically, by means of static analysis and thus, do not require any annotation or manual effort. Furthermore, by detecting them automatically, the potential of changes to the modeled entities and types leading to instability is eliminated.

Another problem with this approach becomes visible when some variables are not in scope, and hence cannot be explicitly mentioned in the specification (Hatcliff et al., 2012). In order to overcome the problem in this context, complex solutions (Reynolds, 1981; O'Hearn, Reynolds, and Yang, 2001; Banerjee, Naumann, and Rosenberg, 2008) based on Hoare logic style frame rules (Hoare, 1971) have been suggested (Hatcliff et al., 2012).

2.3.2 The Exclusive Approach

The most frequent approach to framing is the exclusive approach. This consists in expressing frame properties by means of *modifies-clauses* that list all the variables that may be modified by an operation. Implicitly, everything that is not listed in such clauses is understood as having to remain unchanged (Guttag et al., 1993a). This approach relies on the observation that the *mn* matrix described by McCarthy and Hayes is usually sparse, as most operations affect only a limited number of elements (Meyer, 2015).

Modifies clauses such as ***modifies*** *a, b, c* can be interpreted as a set of clauses of the form $q = \textbf{\textit{old}}(q)$, for any q other than a, b or c. Despite their widely accepted, yet mildly misleading name, a *modifies clause* does not require a command to modify all the listed elements. Essentially, *modifies clauses* put an upper bound on the set of elements that can be modified and imply that it is strictly forbidden to modify *anything else*. The exclusive approach to specifying frame properties owns its name to its characteristic of identifying unaffected elements by exclusion (Meyer, 2015). Bertrand Meyer argues that a more appropriate name for such clauses is *only clauses* (Meyer, 2015), since the main goal is not necessarily to enumerate variables that will change, but rather to specify that everything else, i.e. variables that are not listed, will not change.

This approach has its roots in the *modifies construct* presented by Liskov and Guttag (Liskov and Guttag, 1986). Forms of *modifies clauses* have been used in many different specification languages, including the Larch family (Guttag, Horning, and Wing, 1985; Guttag et al., 1993a), JML (Leavens et al., 2006), Spec# (Mike Barnett, 2005), Dafny (Leino, 2010), and Z (Abrial, Schuman, and Meyer, 1980).

In JML (Leavens, Baker, and Ruby, 2006), *modifies clauses* are called *assignable clauses* and are used for indicating locations that a method may assign to. These are slightly different than classical *modifies clauses* in other languages. For instance, a method assigning to a location a and then re-establishing its original value, is required to list a in its corresponding *assignable clause*. A typical *modifies clause* however, does not require listing a, since the method does not modify a effectively. JML also features *conditional modifies clauses*, allowing methods to specify that a modification may occur only in certain situations. Non-pure methods that do not explicitly specify *assignable clauses* are by default given an *assignable everything* clause. Pure methods have by default an *assignable nothing* clause (Chalin et al., 2005). Additionally, JML provides *accessible clauses* that allow specifying accessed locations (Leavens et al., 2006).

In Dafny (Leino, 2010) *modifies* clauses are expressed by sets of objects and they must be interpreted as giving permissions to a method to modify any field of any object that is a member of the specified set. Frame conditions are thus expressed at the level of objects and not at the level of object fields. While Dafny methods are not required to specify what they read, for Dafny *predicates*, i.e. functions returning Booleans, *reading frame conditions* can also be specified (Koenig and Leino, 2012). These are memory locations that predicates are allowed to read, and they can be specified as sets of objects or object fields. Dafny checks that memory locations outside the reading frame

are not accessed; nested predicate calls must have reading frames that are included in the reading frames of the calling predicate. Predicate parameters are not memory locations and, hence must not be declared. In addition, Dafny uses a form of *dynamic frames* (Kassios, 2006) that we will present in Section 2.4.

In Spec# (Mike Barnett, 2005; Leino and Müller, 2008a), *modifies clauses* can be explicitly added for constraining the modification of objects that were allocated in the pre-state of a method, i.e. new objects allocated and modified by a method need not be included in the *modifies clauses*. Methods can specify that *any* field of an object o may be modified with a construct of the following form ***o.****; it can also be specified that only *some* field a may be modified with a construct of the form ***o.a***. Unlike the clauses expressed using ***old*** in postconditions for excluding some modifications, *modifies clauses* must account for temporary modifications as well (similarly thus to the JML assignable clause interpretation). For instance, for a method decrementing some integer field f and incrementing it subsequently, the method could still specify that $f =$ ***old**(f)* in its postconditions. However, it would also have to include f in its *modifies clause*.

Spec# *implicitly* adds a *modifies clause* to methods in which **this.*** is the only listed element. Thus, by default, methods are allowed to modify any field of the **this** object. To prevent this, the fields that may be modified must be *explicitly* included in the clause (meaning that those not included are not allowed to change). A special construct of the form **this.o** must be explicitly used for specifying that a method does not modify any field of **this** (Leino and Müller, 2008a).

Information hiding imposes mechanisms for abstracting over program state that cannot be explicitly mentioned in the *modifies clause* of a public method. To this end, *wildcards* can be used for specifying that the private representations of objects may be modified, as well as for specifying the modification of state in subclasses (Leino and Müller, 2008a). However, wildcards do not extend to aggregate objects and to this end, Spec# introduces the notion of *ownership* that we will discuss in Section 2.4.

In Boogie, frame conditions are expressed using coarse-grained ***modifies clauses*** in conjunction with postconditions. These can quantify over fields and specify locations of the heap that may be modified (*This is Boogie 2, Boogie Reference Manual*).

SPARK (Barnes and Limited, 1997) uses a variation of the typical exclusive approach. SPARK procedures may reference or update the state associated with their parameters, in addition to that of global variables. SPARK contracts must explicitly account for the global variables accessed (read or written) during procedure execution in a `globals` construct. Additionally, for each parameter or global variable, it must be indicated if it is read only, written only or both read and written. As SPARK is based on the Ada language, this is done by means of *mode annotations*, such as **in, out**, indicating that a parameter or global variable is read only or written only, respectively. The **in out** annotation is used for signaling that the annotated parameter or global variable is both read and written. Together, mode annotations on parameters and globals provide a complete specification of the inputs and outputs of a procedure (Hatcliff et al., 2012). VDM (Jones, 1990) provides similar annotations.

The exclusive convention facilitates the specification of *pure* operations, i.e. operations having no side-effects, on which assertions in various languages, including Eiffel, JML and Spec# rely on for supporting data abstraction. Specifying that an operation is pure simply amounts to specifying an empty ***modifies*** clause. However, specifying and verifying the effects of heap modifications on the results of pure methods has been described as one of the difficult remaining challenges related to framing (Hatcliff et al., 2012).

2.3.3 The Implicit Approach

The implicit approach eliminates the need to specify frame properties per se. One of the implicit approaches relies on limiting what a procedure can modify based on the procedure's precondition. This approach is adopted in separation logic (discussed in Section 2.4) and in the *implicit dynamic frames* (Smans, Jacobs, and Piessens, 2012) technique, where reading and writing to memory requires knowing that the memory contains that location. To this end *accessibility information* is specified in the preconditions of methods. By analysing preconditions, an upper bound on the set of locations that are modifiable by a procedure can be detected. As will be discussed in Chapter 7, our approach to inferring fine-grained modifications can be seen as an implicit one as well. It relies on data-flow analysis and it is entirely automatic, without requiring any dedicated annotations.

Another approach to implicit framing was presented by Meyer. He proposes the inference of frame properties for a method from the method's postcondition (Meyer, 2015). This approach relies on the empirical observation that, in practice, when programmers realize that an element is modified by a method's execution, they will generally include and express information about how the element is modified. It was inspired by an informal review of publicly available JML code, which showed that in practice elements included in an *assignable clause* overlap those appearing in the method's postcondition. Meyer argues that any exception to this observation can be easily addressed by inserting a Boolean function into the postcondition, which always returns ***true*** and which introduces its elements into the implicit frame (Meyer, 2015).

2.4 Topologies and Effects

Specification techniques for complex data structures and operations manipulating them must be able to describe and to address issues related to two different aspects, namely: the topology or structure of the former, and the effects of the latter on the data structures' state (Hatcliff et al., 2012). In the object-oriented realm, objects encapsulate state and functionality, yet their implementations are rarely limited to the fields and methods of a single object. After all, one of the principles of object-oriented programming is to favour *composition* over inheritance. Thus, object fields reference other objects, often of different classes, and those objects in turn, reference yet other objects, and so on. In order to reason about and to prove functional correctness, specifications have to capture this "composite" shape of the implemented data structures (Leino and

Müller, 2008a). They also have to describe the effects of operations on the state of the data structures, including *write effects*, i.e. which parts are potentially modified by an operation, and *read effects*, i.e. which parts are potentially accessed by an operation (Hatcliff et al., 2012).

For objects and heap data structures the write and read effects (Greenhouse and Boyland, 1999) refer to parts of the heap, i.e. locations. Specifications for heap data structures might also require including allocation and deallocation effects, as well as locking information (Hatcliff et al., 2012). Detecting and reasoning about read and write effects is necessary and relevant in different situations. For instance, Greenhouse and Boyland (Greenhouse and Boyland, 1999) present an effects system for performing semantics-preserving program manipulation on Java source code.

Our work is done in the context of a purely functional language, with immutable data structures and no destructive updates. Reasoning about the heap is beyond our scope. However, our concerns are similar: we handle "composite" data structures modeled by immutable associative arrays and algebraic data types, i.e. structures and variants, and we want to capture the behaviour of operations receiving such a composite input, manipulating it, reconstructing it and returning its *new state* into a composite output. Thus, in contrast to specification and reasoning techniques for objects which are concerned with deep-heap effects, we are concerned with deep-state effects.

Specification techniques for topologies and effects must address three major challenges, namely abstraction, reasoning and framing (Hatcliff et al., 2012).

Abstraction. In the object-oriented context, heap properties must be expressed in an implementation-independent manner. Abstraction is important for information hiding and for supporting subtyping (Leino, 1998; Leavens and Müller, 2007). Aspects related to visibility and information-hiding are orthogonal to our work. The language we are working with does not have subtyping. Therefore, disclosing the topology of our data structures is not problematic from this point of view.

Reasoning. The formal framework in which (heap) properties are expressed should allow efficient, ideally automatic reasoning.

Framing. Specifications of heap operations should ease reasoning about framing and aid in proving that certain heap properties are not affected by a heap operation. Framing can be illustrated by the following rule, expressing that a state that is unmodified by C can be preserved:

$$\frac{\{P\}C\{Q\}}{\{P \wedge R\}C\{Q \wedge R\}}$$

if the write effect of C is disjoint from the free variables of R. In the presence of complex heap data structures, the disjointness of the effects of C and the assertion R is more difficult to express, as it needs to specify that the locations that are modified by C are disjoint from the locations read by R. Similarly, though not referring to locations, we

have to be able to express that the substructures (or subelements) modified by C and those read by R are disjoint.

The sets of written or read locations are called *footprints*. Hatcliff et al. classify approaches to the specification of heap properties into three categories. The first category relies on *explicit footprints* and uses sets of objects or locations that are included in predicates and effects specifications. Dynamic frames (Kassios, 2006; Kassios, 2011) and region logic (Banerjee, Barnett, and Naumann, 2008; Banerjee, Naumann, and Rosenberg, 2013) are the main exponents of this category. The second category relies on *implicit footprints*, which are derived from predicates in specialized logics, such as separation logic. The third approach relies on *predefined footprints*, which are derived from predefined heap topologies (Hatcliff et al., 2012). Ownership types (Clarke, Potter, and Noble, 1998) are the main exponent of this category. All of these techniques allow specifying the topologies of common heap data structures and reasoning about the effects of operations. However, each amounts to a different balance between expressiveness and automation (Hatcliff et al., 2012).

2.4.1 Explicit Footprints

The explicit footprint approach to framing was pioneered by Kassios and the dynamic frame theory (Kassios, 2006; Kassios, 2011). This proposed adding sets of locations to the specification language and expressing footprints in terms of such sets. For preserving information hiding, these sets of locations can involve *dynamic frames*, specification variables that abstract over a set of locations. The initial solution based on dynamic frames was formalized in the context of an idealized logical framework, using higher-order logic and inductive-based proofs which are difficult to automate. Subsequent work on region logic (Banerjee, Naumann, and Rosenberg, 2008; Banerjee, Barnett, and Naumann, 2008; Banerjee, Naumann, and Rosenberg, 2013) and the Dafny verifier on one hand, and VeriCool (Smans, Jacobs, and Piessens, 2008) on the other hand, developed dynamic frames in a first-order setting.

VeriCool uses *pure methods* for describing sets of locations. Recursively defined pure methods or logic functions can be a challenge for automatic theorem provers (Hatcliff et al., 2012; Banerjee, Barnett, and Naumann, 2008).

In region logic, for minimizing the need for inductively defined predicates in specifications, the specification attributes used in the dynamic frames approach (Kassios, 2006) are replaced with *ghost state* (Banerjee, Naumann, and Rosenberg, 2013), i.e. mutable auxiliary fields and variables. Programs have to be explicitly annotated with these, which might imply a cumbersome manual effort but, unlike the dynamic frame theory in its original form, this permits automated theorem proving.

Zee et al. have used explicit footprints for verifying the functional correctness of linked data structures in Jahob (Zee, Kuncak, and Rinard, 2008). Banerjee et al. (Banerjee, Naumann, and Rosenberg, 2008; Banerjee, Barnett, and Naumann, 2008) encoded region logic in the intermediate verification language Boogie (Leino and Rümmer, 2010).

The dynamic frames approach using ghost variables is supported by the Dafny language (Leino, 2010; Koenig and Leino, 2012). As described in Section 2.3.2, Dafny supports the exclusive approach to specifying frames. Ghost variables are used in **modifies** clauses. The standard idiom consists in declaring a set-valued ghost field, *Repr* for instance, to dynamically maintain *Repr* (i.e. explicitly update it in the code) as the set of objects that are part of the receiver's representation, and to use *Repr* in **modifies** clauses (Leino, 2010). The following idiom is standard (Leino, 2010):

```
class MyClass {
  ghost var Repr: set<object>;
  method SomeMethod() modifies Repr; { /*...*/ }
}
```

This ***modifies*** clause is to be interpreted as: the method may modify any field of any object in *Repr*. If **this** is a member of the *Repr* set, then the ***modifies*** clause also allows the method to modify the field *Repr* itself (Leino, 2010).

With explicit footprints, proving frame properties consists in proving that the *read effects* of a predicate and the *write effects* of a method are disjoint.

Before the dynamic frame approach, data groups (Leino, 1998; Leino, Poetzsch-Heffter, and Zhou, 2002) and solutions based on the *Universe type system* (Müller, 2002) have been proposed for specifying footprints within single objects.

The level of expressiveness offered by techniques based on explicit footprints is very high, allowing specifications to relate different regions in arbitrary ways, ranging from disjointness or inclusion of regions to characterizing their intersection. However, this flexibility complicates reasoning. When regions are stored explicitly in ghost variables as is done in Dafny, programs need to explicitly update these ghost variables to maintain invariants. This can prove to be a cumbersome task. When pure methods are used as in VeriCool, it is mandatory to reason explicitly about the effects of heap modifications on their results (Hatcliff et al., 2012).

2.4.2 Implicit Footprints

The implicit footprint approaches rely on specialized logics for implicitly representing footprints. Separation logic (O'Hearn, Reynolds, and Yang, 2001; O'Hearn, Yang, and Reynolds, 2004; Reynolds, 2002; Reynolds, 2005; Reynolds, 2000) is the most prominent representative of this category.

Separation logic extends Hoare logic (Hoare, 1971) with the *separating conjunction operator* $*$. Each assertion in separation logic defines a portion of the heap. The assertion $P * Q$ is true if and only if P and Q hold for disjoint parts of the heap. Local reasoning is fundamental to separation logic (O'Hearn, Reynolds, and Yang, 2001); specifications need to describe all the state that the code C reads or writes. Thus, in the triple $\{P\}C\{Q\}$, P must be interpreted as being all the state that is needed for executing C, i.e. the footprint of C. This interpretation of Hoare triples leads to the following frame rule in separation logic:

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}$$

which allows inferring that a local property is preserved for a wider state, obtained by extending P with another disjoint state R. Some versions of separation logic impose additional conditions about local variable modifications as the $*$ operator only separates heaps. Separation logic can be extended such that $*$ also separates variables, thus eliminating the need for additional conditions (Parkinson, Bornat, and Calcagno, 2006).

A separation logic for Java was introduced by Parkinson (Parkinson and Bierman, 2005). This has primitive assertions to describe the values of fields in the heap and allows describing portions of the heap containing several disjoint objects using the $*$ operator.

Separation logic does not require explicitly specifying read or write effects. They are implicit in a method's precondition. Data structures are specified using logic functions. By including such a logic function in a method's precondition, the method is allowed to read and write anything belonging to the footprint of the logic function, but cannot access anything outside this footprint.

Approaches based on separation logic are hard to implement and to integrate into verification tools. Verifiers based on separation logic have mostly relied on symbolic execution and have not yet achieved the same level of automation as verifiers based on verification condition generation (Hatcliff et al., 2012). However, currently a series of tools exist that can reason using separation logic. These include Smallfoot (Berdine, Calcagno, and O'Hearn, 2005; Berdine, Calcagno, and O'Hearn, 2012), SpaceInvader (Distefano, O'Hearn, and Yang, 2006; Calcagno et al., 2008), jStar (Distefano and Parkinson, 2008; Naudziuniene et al., 2011), VeriFast (Jacobs, Smans, and Piessens, 2010; Jacobs et al., 2011) and SLAyer (Berdine, Cook, and Ishtiaq, 2011).

The *implicit dynamic frames* approach (Smans, Jacobs, and Piessens, 2012) unifies the dynamic frames concept with separation logic. Framing specifications of a method are inferred using an implicit approach, as described in Section 2.3.3. They are encoded in first-order logic and can be used for automatic verification with SMT solvers. This is done in VeriCool (Smans, Jacobs, and Piessens, 2008) and Chalice (Leino, Müller, and Smans, 2009).

2.4.3 Predefined Footprints

In contrast to the implicit and explicit footprint approaches which describe properties found in a program, the third approach focuses on reasoning efficiently about programs with restricted topologies. Ownership types (Clarke, Potter, and Noble, 1998) are representative of this approach.

Ownership types typically enforce a tree topology, whereby every object in the heap has at most one owner object and the owner relation is acyclic. Topological properties beyond this tree structure have to be expressed using object invariants and predicate logic. Read and write effects typically use ownership as an abstraction mechanism: the

right to read or write an object include the right to read or write all the objects it (transitively) owns (Hatcliff et al., 2012).

Spec# addresses framing through ownership types: without explicit specifications stating otherwise (***modifies*** clauses of the form presented in Section 2.3.2), methods may modify only the fields of the receiver and of those objects within the subtree of which the receiver is the root. Ownership is expressed by means of attributes on field declarations (Barnett et al., 2004; Barnett et al., 2011).

Ownership has been used to verify write effects (Müller, Poetzsch-Heffter, and Leavens, 2003) and invariants (Drossopoulou et al., 2008; Leino and Müller, 2004; Müller, Poetzsch-Heffter, and Leavens, 2006). All the existing ownership-based verification techniques enforce that all modifications of an object must be initiated by the object's owner. This gives owners total control over modifications of their internal representations and allows them to maintain invariants (Hatcliff et al., 2012). Ownership-based approaches have been used for reasoning about model fields (Leino and Müller, 2006) and for enforcing object immutability (Leino, Müller, and Wallenburg, 2008).

The ownership topology can be enforced by type systems (Lu, Potter, and Xue, 2007; Müller, 2002). In JML it is enforced through universe types (Dietl and Müller, 2005). In Spec# it is encoded as object invariants (Barnett et al., 2004).

Reasoning about framing relies on the tree structure on the heap enforced by ownership. The ownership trees rooted in two different objects o_1 and o_2 are disjoint if neither o_1 owns o_2, nor o_2 owns o_1. The disjointness of ownership trees can then be used to prove that read and write effects of methods do not overlap (Hatcliff et al., 2012).

2.5 Other Approaches to Reason about Frames

Rakamarić and Hu report in (Rakamaric and Hu, 2008) a method to infer frame axioms of procedures and loops based on static analysis. As a starting point, they use the DSA shape analysis, presented by Lattner et al. (Lattner, Lenharth, and Adve, 2007). DSA provides a summary of points-to relations as a graph, that is used to compute a set of memory locations that are modified by a procedure or its callees. By a pass through the graph, for each node that is reachable from the globals or procedure parameters, they generate expressions representing a path to that node. The generated frame axioms are used internally by an extended static checker of C programs, i.e. in a purely automatic setting.

In (Taghdiri, Seater, and Jackson, 2006), Taghdiri et al. present a technique for extracting procedure summaries for object-oriented procedures, used to prove verification conditions. Procedures are executed symbolically and the environment of the post-state is computed so as to express every variable and field in terms of the values of the variables and fields of the pre-state. The extracted procedure summaries can be viewed as detailed frame conditions, describing which memory locations might be changed and how.

In (Sozeau, 2009), Sozeau presents a generalized rewriting technique implemented in the Coq proof assistant that allows substituting a term t of an expression by another term t' when t and t' are related by a relation R. This generalizes equational reasoning to reasoning modulo arbitrary relations. The technique relies on dependent types and is based on a constraint generation algorithm generating type class constraints. The Coq tactic supports polymorphic relations, morphisms and subrelations.

Bertrand Meyer proposed the *double frame inference* strategy, an approach that targets the automation of both frame specification and frame verification in the context of Eiffel (Meyer, 1991), an object-oriented language with native support of *Design by Contract* features (Meyer, 1992). The first component – the frame specification inference – relies on the analysis of method postconditions as described in Section 2.3.3 and obtaining a set $\bar{p}$. This represents an overapproximation of the set of elements that are allowed to be modified by p according to its specification. The second component of the strategy, *the frame implementation inference* relies on the frame calculus (Kogtenkov, Meyer, and Velder, 2015), which is itself based on alias calculus (Kogtenkov, Meyer, and Velder, 2015; Meyer, 2010; Meyer, 2011). Methods are analysed and $\underline{p}$ is detected; this represents an overapproximation of the set of expressions whose values may change as a result of executing p. Frame verification amounts to verifying that $\bar{p}$ includes $\underline{p}$.

2.6 Other Relevant Work

Pure methods, also known as queries or observers, are side-effect free methods that always evaluate to the same result value given the same input value. They are intensively used for providing specifications for methods without disclosing implementation details in languages such as JML, Spec# and Eiffel. Leavens et al. identify the development of specification and verification techniques for determining the effects of heap modifications on the results of pure methods as one of the remaining challenging problems related to framing (Leavens, Leino, and Müller, 2007). Though our work is not concerned with heap modifications, we are interested in the *dependency* of Boolean Smart predicates, i.e. logical properties, on the layered ("composite") data structures they are receiving as inputs. In Chapter 5 we present a static analysis meant to capture such dependencies.

Various encodings of pure methods (Cok, 2005; Darvas and Müller, 2006) in program logic have been proposed, but they do not cover aspects related to reasoning about frame properties when the specifications make use of pure methods. Some specification techniques for frame properties (Leavens, Baker, and Ruby, 2006; Leino and Müller, 2006; Leino and Nelson, 2002; Müller, Poetzsch-Heffter, and Leavens, 2003) allow describing the fields that are potentially modified by a method execution using ***modifies*** clauses. These however do not specify the effects of a method execution on the results of pure methods (Leavens, Leino, and Müller, 2007).

One technique for determining the effects of heap modifications on the results of pure methods requires listing all pure methods that are potentially affected by a method, in the method's ***modifies*** clause. This approach is adopted in COLD-K (Feijs and

Jonkers, 1992), where the frame of a procedure specification lists the variables and the equivalent of pure methods whose value may be changed by the procedure. For dealing with modularity issues, COLD-K also makes use of *read effects*.

Other approaches (Leino and Müller, 2006; Müller, Poetzsch-Heffter, and Leavens, 2003) for determining effects on the results of pure methods rely on *model fields*. These are specification-only constructs, whose value is determined by applying a mapping to the concrete state of an object. They are similar to pure methods, but unlike the latter, they do not have parameters and they are required to be *confined* (Leino and Müller, 2006; Müller, Poetzsch-Heffter, and Leavens, 2003).

Approaches based on model fields require that pure methods read only the state of the receiver object and its sub-objects. This information about the *read effect* of a pure method can be used to determine which *write effects* potentially have an impact on the result of a pure method. In general, it can be proven that a method m does not affect the result of a pure method p if the write effect of m and the read effect of p are disjoint (Leavens, Leino, and Müller, 2007).

There are various approaches to using read effects for reasoning about pure methods. One approach relies on complete specifications of result values included in the postconditions of pure methods. Used in conjunction with ***modifies*** clauses, these allow determining whether a method affects the result of a pure method (Leavens, Leino, and Müller, 2007). Various solutions based on explicitly specified read effects exist (Feijs and Jonkers, 1992; Greenhouse and Boyland, 1999; Jacobs and Piessens, 2006). Specification of these using data groups (Leino, 1998; Leino, Poetzsch-Heffter, and Zhou, 2002) and an effects system built on top of an ownership type system (Clarke and Drossopoulou, 2002) have been proposed. Multi-threaded programs also require such specifications (Praun and Gross, 2003).

Chapter 3

The Smart Language and ProvenTools

> Languages are not strangers to one another.
>
> Walter Benjamin

In this chapter, we introduce Smart, a programming and specification language developed at Prove & Run, as well as the toolchain associated with it. While not claiming to be exhaustive, we give an overview of the language's features and syntax in Section 3.1. In Section 3.2, we present the tools manipulating Smart models. Section 3.3 briefly presents Smil, the Smart *Intermediate Language*. A computational version of it – αSmil – is targeted by the static analyses presented throughout the remainder of this thesis. The following chapter will focus entirely on αSmil, illustrating its usage and introducing its syntax and formal semantics.

3.1 The Smart Modeling Language

Smart is a modeling language developed at Prove & Run. It constitutes a unified programming and specification language, designed to facilitate proofs. One of the common, often cited reasons why programmers reject the use of formal methods is that they are not willing to learn a separate language just for specifying their programs, in particular if that language is fundamentally different from the programming language. Smart addresses this issue by allowing one to both develop the implementation of programs and to specify their logical properties in a single language.

The Smart language is a purely functional (side-effect free), strongly-typed, polymorphic first-order language. The basic building blocks of programs written in Smart are *predicates*, the equivalent of functions in other common programming languages. Besides the common primitive types that are traditionally available as built-in types, algebraic data types (structures and variants), and associative arrays are provided as well. *Exit labels* constitute the language's main specificity; they facilitate separating data- and control-flow in programs.

In addition, being designed in order to write code that will subsequently be proven, the language allows the definition of various types of logical specifications as well.

These range from pre- and postcondition contracts, local assertions and loop invariants to inductive predicates, lemmas and hypotheses.

ProvenTools is a complex set of development tools for the Smart language. It has been developed at Prove & Run with the goal of facilitating the achievement of high-level certifications. The toolchain has the structure of a set of Eclipse plug-ins of JDT type – Java Development Tools (*Eclipse Java Development Tools (JDT)*). Together, these constitute a complete Integrated Development Environment (IDE) allowing one to not only write, edit and document Smart models, but also to browse proof obligations, to prove them by employing a built-in prover and finally, to generate executable code in C.

ProvenCore[1] (Lescuyer, 2015) and *ProvenCore-M*[2] are two microkernels that have been completely modeled in Smart and developed using ProvenTools. The former is a general-purpose microkernel that ensures isolation, i.e. integrity and confidentiality. The latter targets embedded devices based on microcontrollers.

Throughout the rest of this section we will present some of the main concepts and mechanisms of Smart, discussing predicates, control flow, algebraic data types and specification-only constructs.

3.1.1 Smart Predicates and Types

Smart supports modular program development with a straightforward module concept. *Modules* constitute the compilation units of Smart programs and any valid Smart program consists of a non-empty set of modules, which are themselves organized in *packages*. Modules have an identifier that is unique in each program and, in practical terms, each module corresponds to a file. Modules can import other modules and they contain a list of type and constant declarations, as well as a list of *predicates*.

Predicates, the equivalent of functions in other common programming languages, are the basic building blocks of programs written in Smart. Though named in reference to *predicate logic*, predicates in Smart receive a number of inputs and produce a number of outputs in return, in contrast to predicates in mathematics, which are commonly understood to be Boolean-valued functions of the form:

$$P : X \to \{true, false\}.$$

Smart predicates can be classified in two different categories, namely *implicit* and *explicit* predicates, based on their implementation or their lack thereof.

Implicit predicates can be seen as a form of an *assumption*: as their names suggest, they are not implemented *per se*, but simply declared using the **implicit program** keywords. Such predicates are similar to the declarations of native methods in Java or external functions in C. Traditionally, in Java, programmers use the *Java Native Interface* (JNI) (Liang, 1999; *Java Native Interface Documentation (JNI)* 1999) when they need to implement small, time-critical code portions in a lower-level language,

[1] http://www.provenrun.com/products/provencore/
[2] http://www.provenrun.com/products/provencore-m/

such as assembly, or when they need to access a library already written in another programming language such as C. In Smart, implicit predicates play an important role with respect to code documentation. Their implementation is not provided in the model, but, as we will further explain in Section 3.1.4, they can be used to specify logical properties of the explicit implementations provided externally in a lower-level language, typically in C or assembly.

For example, an implicit predicate converting an integer given as an input into a float can be declared as follows:

```
public float_of(int n, float f+)
implicit program
```

The predicate's result is given a name, `f`, and it is introduced as one of the predicate's parameters. It is marked as being the predicate's output by the `+` symbol following it and is thereby syntactically distinguished from the predicate's input parameter, `n`, which is unadorned.

In the general case, Smart predicates can have any number of input or output parameters. However, a parameter cannot be both at the same time and each of these must be explicitly marked either as an input or as an output. An input parameter's value can be read and used in the predicate's implementation. An output parameter's value must be constructed by the predicate's implementation and returned as a result. Furthermore, values in Smart are *immutable*. As a consequence, Smart predicates are *pure*: it is impossible to pass a parameter "by reference" and modify a predicate's input as a side-effect. Smart is thus a side-effect free language which provides *referential transparency* (Strachey, 1967). Furthermore, the language supports neither global variables, nor global states, but can be characterized rather as a *state-passing style* language. Smart predicates are *deterministic*: they always return the same output any time they are called with a specific set of input values. In particular, this is a prerequisite for implicit predicates.

As mentioned in the introduction, Smart is also a *strongly-typed* language. Each input and output parameter of a predicate must have an associated type and the usage of an object of some type where a parameter of another data type is expected is forbidden by the language. Unsafe conversions between different types are forbidden as well. Smart provides various built-in types, such as int, short, long, char, boolean, float and double, that are traditionally available in other programming languages as well. Additionally, users can declare new types with the **type** keyword and then define predicates manipulating these types. As in the case of predicates, *implicit data types* can be simply declared without being explicitly defined. For example, supposing that an implicit data type called `cartesian_point` and the predicates manipulating it, are defined in a lower-level language, we would make them available to other Smart predicates using the following declarations:

```
// Implicit data type declaration
type cartesian_point
```

```
/* Retrieve coordinate on X-axis. */
public get_X(cartesian_point p, float x+)
implicit program

/* Retrieve coordinate on Y-axis.*/
public get_Y(cartesian_point p, float y+)
implicit program

// Construct a new point p with coordinates (x, y).
public new_point(float x, float y, cartesian_point p+)
implicit program

//Pretty-print
public print_point(cartesian_point p)
implicit program
```

Some implicit predicates manipulating inputs of type `cartesian_point` are declared as well: the first two of them – `get_X` and `get_Y` – simply return the input point's numerical coordinates on each of the Cartesian system's axes. The next predicate, `new_point` creates and returns a new point from the two given input coordinates. Alternatively, it is possible to directly declare and implement these types and predicates in Smart as we will show in the following paragraphs. The last one, `print_point` simply displays the input point without effectively producing an output. As shown in the example, similarly to Java, *comments* in Smart can be introduced by using `//` for single-line comments or `/* */` for multi-line comments. Similarly to Javadoc, code documentation can be given using the begin-comment delimiter `/**`.

In general, implicit data types and the implicit predicates manipulating them can act as a public interface for a concrete class, showing the type and the operations allowed to manipulate values of that type, but hiding the implementation.

Explicit data types can be declared and defined using structures and variants. For example, we could explicitly define the type `cart_point` by means of a structure having two different *fields* of type **float**, called `x` and `y`. Each of them corresponds to the point's numerical coordinates on the *X*- or *Y-axis*, respectively:

```
type cart_point = {
  float x;
  float y;
}
```

For representing a point in a polar coordinate system we can define a different type `polar_point` as follows:

```
type polar_point = {
  /* Radial coordinate (distance from the pole) */
  float radius;
```

```
    /* Polar angle */
    float azimuth;
}
```

Explicit predicates have explicitly defined implementations, following immediately after their declaration, which strongly resembles that of an implicit predicate but from which the keyword **implicit** is omitted. Their bodies are sequences of several statements, which are essentially calls to other predicates. For example, to *translate* a point (x, y), i.e. to add a given pair of numbers (a, b) to its Cartesian coordinates and obtain the new point $(x', y') = (x + a, y + b)$, a predicate `translate_point` could be defined in the following manner:

```
/* Convert x to float, add it to y and retrieve the sum. */
public sum_of(int x, float y, float s+)
implicit program
public translate_point(cartesian_point p, int a,
                       int b, cartesian_point q+)
program {{ float xa, float yb }} // Local variables
  {
    print_point(p);          /* 1 */

    get_X(p, xa+);           /* 2 */
    get_Y(p, yb+);           /* 3 */

    sum_of(a, xa, xa+);      /* 4 */
    sum_of(b, yb, yb+);      /* 5 */

    new_point(xa, yb, q+); /* 6 */

    print_point(p);          /* 7 */

}
```

The body of the `translate_point` predicate consists in a sequence of several statements: the first of these simply pretty-prints the input point `p`. The next two statements are calls to accessors of `p`'s coordinates on the X- and Y-axis, which are stored in the local variables `xa` and `yb`, respectively. Next, the coordinates $(xa', yb') = (a+xa, b+yb)$ for the translated point are computed by calling the `sum_of` predicate, which returns the float sum of an integer and a float. The output point `q` is constructed by calling the constructor `new_point` with `xa` and `yb` as inputs. The last statement pretty-prints the input point `p` again.

As illustrated by our example, each call to a predicate is made by passing the parameters in the same order as in the predicate's declaration and by explicitly marking any output with `+`[3]. Replacing line 4 with `sum_of(xa, a, xa+)` would result in an

[3]This is mandatory because of overloading.

error, because the first input parameter of a call to `sum_of` is expected to be an integer and the second, a float. Similarly, omitting the `+` symbol at line 6 and writing `new_point(xa, yb, q)` would result in an error. By explicitly marking the outputs of each statement, it is straightforward to distinguish between the variables that are actually written by the statement and those that are used only as inputs. Furthermore, since predicates are not allowed to modify their inputs, the language strictly forbids using a predicate's input parameter as an output for any statement in the predicate's body. Thus, in our example predicate, we are prevented from using the input point `p` as the output of the `new_point` predicate call. However, outputs and local variables, such as `xa` and `xb`, can be written to but reading them (i.e. using them as inputs for a predicate call) before they have been written at least once, amounts to using uninitialized variables and behaves in an unspecified manner. In our example, `xa` and `ya` are used as both inputs and outputs at line 4 and 5, respectively. This is correct since `xa` and `ya` are local variables that have already been written to by the statements at line 2 and 3 preceding the calls to `sum_of`.

We stress again the fact that destructive updates are not possible in Smart: even if at a first glance a statement such as the call to `sum_of` at line 4 might give the impression that `xa` is modified in place, all that the statement actually does is to create a new **float** whose value is obtained by adding the old value of `xa` to the value of `a` and then to set `xa` to reference this new **float** instead of the old one. A simple conversion to a *static single assignment form* (Cytron et al., 1989) would eliminate these assignments and show the absence of any mutation whatsoever. Thus, were we to inspect the state of the input point `p` before and after the calls to `sum_of`, we would observe that it remains unchanged: this is what we do when printing `p` again at the end of `sum_of`.

As a last remark about our example, it is noteworthy to mention that the statement `new_point(xa, yb, q+)` which produces the predicate's output, is not the predicate's last statement. Smart does not support any dedicated *return* statement. Instead, when exiting from a predicate, the outputs hold the values that they have been assigned when executing the body. This mechanism allows one to define predicates having multiple outputs. Their names are chosen by the programmer and their values can be modified multiple times during the predicate's execution; however, the values retrieved are the ones that are available at the moment the program exits the predicate.

3.1.2 Exit Labels and Control Flow

Besides input and output parameters, the declaration of a predicate can also include a set of *exit labels*. When called, a predicate exits with one of the specified exit labels, thus summarising and returning to its callers further information regarding its execution.

Exit labels constitute the main specificity of the Smart language. They can denote different exceptional execution scenarios and act as exit codes, similarly to exceptions and exit status return values in other programming languages.

Every predicate has a non-empty set of labels; by default, any predicate has the built-in exit label **true** that denotes the successful exit status of a predicate. The predicates illustrated previously in Section 3.1.1 did not have explicitly declared exit

labels; in such a case, it is assumed that the only possible exit label for the predicate is **true**, and hence, that the predicate will succeed in all circumstances.

Returning to our previous example, the predicate `translate_point`, we could have written its complete declaration by explicitly stating that **true** is the only possible exit label:

```
public translate_point(cartesian_point p, int a,
                       int b, cartesian_point q+)
-> [true]
program {{...}} {
  ...
 }
```

This declaration is strictly equivalent to the one given in Section 3.1.1.

In the general case, any number of labels can be specified after the parameters. For example, we could declare a predicate that converts the coordinates of an input point (x, y) of type `cartesian_point` to polar coordinates:

$$r = \sqrt{x^2 + y^2}$$
$$\phi = atan2(y, x)$$

and returns a point (r, ϕ) of type `polar_point` with these coordinates. For computing the second polar coordinate, the *polar angle* or *azimuth*, the predicate would call another predicate *atan2*, which is the arctangent function with two arguments, a common variation on the arctangent function. The *atan2* function avoids the problem of division by zero; however it is undefined when both x and y, i.e. the Cartesian coordinates, are zero. For declaring it in Smart we can add a special exit label for the case when the given input coordinates represent the *origin* and the result cannot be returned:

```
/* Computes atan(y/x) */
public atan2(float x, float y, float at+) -> [true, undef]
implicit program
```

The declared label's name, `undef`, is a custom name and any valid identifier can be chosen and used as a label in Smart. As previously mentioned, the exit label **true** is predefined and has a special meaning. Another predefined label that is interpreted in a special manner by conditional statements and logical operators is the **false** label. Together, these two exit labels offer a convenient manner to model a Boolean result. Frequently, a Boolean output value can be replaced by declaring these two possible exit labels: **true** to denote a successful execution of the predicate and **false**, respectively.

Besides indicating the followed execution scenario, exit labels play an important role with respect to control flow management. Primarily, the exit label of a call to a predicate determines whether the next predicate call in sequential order should be executed or not: when the predicate exits with **true**, the program can proceed to the

next statement in the program. Any other exit label `lbl` disrupts the normal control flow and forces the current predicate to exit with label `lbl`.

For example, a predicate `cart_to_polar` can be defined with two exit labels, **true** and `undef` as well. It takes two **float** numbers, `x` and `y`, computes the corresponding polar coordinates, `r` and `phi`, by calling the predicates `compute_radius` and `atan2`, and constructs a new point `p` of type `polar_point` using the computed values:

```
public compute_radius(float x, float y, float r+)
-> [true]
implicit program

public cart_to_polar(float x, float y, polar_point p+)
-> [true, undef]
program {{ float phi, float r }}
 {
  compute_radius(x, y, r+);
  atan2(y, x, phi+);
  new_polar_point(r, phi, p+);
 }
```

There is no guarantee that the call to `atan2` will return successfully with exit label **true**: it might return with `undef`, in which case the execution of `cart_to_polar` will break at that point and exit with label `undef`. Furthermore, no output will be generated. In Smart, exit labels condition the existence of output parameters: every output is associated to an exit label `lbl` and it is generated if and only if the predicate exits with that particular exit label `lbl`. All other outputs are discarded and can be considered as unchanged by the caller. The same output can be associated to multiple labels. By default, if no output parameters are specified for a label, it means that no outputs are generated when the predicate exits with this label. The only exception to this rule is made in the case of the built-in **true** label: since **true** normally represents a successful execution, every output of the predicate is associated to it by default. For example, the previous declaration of `cart_to_polar` is strictly equivalent to:

```
public cart_to_polar(float x, float y, polar_point p+)
-> [true: <p>, undef: <>]
program {{ float phi, float r }}
 {
  compute_radius(x, y, r+);
  atan2(y, x, phi+);
  new_polar_point(r, phi, p+);
 }
```

Exit labels can thus behave similarly to exceptions in other programming languages. In order to handle specific observed execution scenarios, Smart provides *label transformers*, which allow catching labels before they escape the current predicate and transforming

them into another label. Complex control flow can be expressed by indicating a set of rules of the form `lbl1 : lbl2` whose role is to transform the label `lbl1` into `lbl2` and by associating them to statements.

For example, we could let the predicate `cart_to_polar` return the label `origin_fail` when the inner computation of the azimuth fails, instead of just forwarding the label returned by `atan2`:

```
public cart_to_polar(float x, float y, polar_point p+)
-> [true: <p>, origin_fail]
program {{ float phi, float r }}
 {
  compute_radius(x, y, r+);
  [undef : origin_fail]
    atan2(y, x, phi+);
  new_polar_point(r, phi, p+);
 }
```

Alternatively, we could also handle the failure of the computation by using transformers and constructing the output point differently: for example, by declaring a *constant* representing the azimuth of the origin, often called *pole* in polar coordinates, and using this for the construction of `p` when the call to `atan2` fails.

```
public const float POLEAZIMUTH;

public cart_to_polar (float x, float y, polar_point p+)
-> [true : <p>]
program {{ float phi, float r }}
{
  compute_radius (x, y, r+);
  [done : true]{
    [true : done, undef : true]
    atan2(y, x, phi+);
    phi := POLEAZIMUTH;
  }
  new_polar_point (r, phi, p+);
}
```

In the following, we show how the control flows when `atan2` terminates with label **true**. The green arrows indicate how control is passed from one statement to the other, based on their exit labels, when starting from the call to the `atan2` predicate:

```
public const float POLEAZIMUTH;
public cart_to_polar (float x, float y, polar_point p+)
-> [true : <p>]
program {{ float phi , float r }}
{
  compute_radius (x, y, r+);
  [done : true]{
    [true : done, undef : true]
    atan2(y, x, phi+);
    phi := POLEAZIMUTH;
  }
  new_polar_point (r, phi, p+);
}
```

And here is how the control flows when `atan2` terminates with label `undef`:

```
public const float POLEAZIMUTH;
public cart_to_polar(float x, float y, polar_point p+)
-> [true : <p>]
program {{ float phi , float r }}
{
  compute_radius (x, y, r+); /* 1 */
  [done : true ]{/* 2 */
    [true : done, undef : true]/* 3 */
    atan2(y, x, phi+); /* 4 */
    phi := POLEAZIMUTH; /* 5 */
  }
  new_polar_point (r, phi, p+);
}
```

After computing the radius `r` by calling `compute_radius`, this new version of the predicate starts by calling the predicate `atan2`. If this operation succeeds, then `phi` is the value of the azimuth and we can use this value as the second input parameter for the point's constructor, `new_polar_point`. This is done by transforming **true** to a new label `done`, whose effect is to jump immediately to the outer block, in this case the top-level. The top-level block of the program catches `done`, transforms it back to **true** and continues with the statement following the block, namely `new_polar_point`, which will construct the output `p`, by using `r` and `phi`, the value of the azimuth returned by `atan2`. When `atan2` is undefined, the transformer `undef : true` is used to jump to an additional statement: `phi := POLEAZIMUTH`, that assigns the value of `POLEAZIMUTH` to `phi`. The constructor is reached in this case as well. However, this time the value of `phi` written at line 5 is used as the second input parameter. We note that the statement at line 5 is a call to a built-in assignment predicate, denoted by `:=` and using an infix notation.

The constant `POLEAZIMUTH` is declared using the keyword **const**. In Smart, constants can be declared and used directly as inputs for predicate calls.

In the general case, arbitrarily complex control flows can be expressed by coupling label transformers, blocks and recursion.

In order to facilitate the user's task of simulating common control flow structures with labels and transformers, Smart provides various control flow statements, which are themselves based on this mechanism. These include a construct that is equivalent to the **try** ... **catch** ... mechanism in Java, a conditional **if** ...**then** ...**else** control structure, as well as the common logical operators for negation (`!`), conjunction (`&&`), disjunction (`||`), implication (`=>`) and equivalence (`<=>`).

Given the Cartesian coordinates (x, y), the first polar coordinate, the radius, is obtained by computing:

$$\sqrt{x^2 + y^2}.$$

For explicitly defining the predicate `compute_radius`, we would first need to implement a predicate `sqrt`, computing the square root of a given positive number. Such a predicate can be recursively implemented as follows, by using the **if** ...**then** ...**else** construct and three implicit predicates:

```
/* Newton-Raphson Square Roots Finding Algorithm */

/* Divides a to b and retrieves result in div. */
public div_double(double a, double b, double div+)
-> [true, undef]
implicit program

/* Check if a is close enough to b: |a - b| < b * 0.001 */
public close_approximation(double a, double b)
-> [true, false]
implicit program

/* Compute ((b + a/b) / 2) */
public better_approximation(double a, double b, double g+)
-> [true, undef]
implicit program

public sqrt(double x, double g, double sqr+)
-> [true, undef]
/* Returns the square root of x by making recursive calls
with better and better guesses g, until reaching a guess
that is close enough to the actual square root's value. */
program {{ double aux }}
{
  div_double(x, g, aux+);
  if close_approximation(aux, g);
  then {
    sqr := g;
  }
```

```
    else {
      better_approximation(x, g, aux+);
      sqrt(x, aux, sqr+);
    }
  }
```

Besides recursion, Smart also supports loops by providing a specific construct that is similar to a traditional "while" loop in other programming languages:

```
while {
...
}
```

The body of this **while** block is repeatedly executed until a dedicated exit label called **exit** tries to escape, in which case the loop is aborted and the execution continues after the block. A "break" can be achieved by *raising* the special **exit** label inside the loop.

For instance, the previously recursive predicate `sqrt`, can be implemented iteratively with a **while** loop as follows:

```
public sqrt_iter(double x, double g, double sqr+)
-> [true, undef]
/* Computes the square root of x iteratively. */
program
 {
  div_double(x, g, sqr+);
  while {{ double aux }} {
   [true : exit, false : true]
     close_approximation (sqr, g);
   better_approximation(x, g, aux+);
   div_double(x, aux, sqr+);
  }
 }
```

3.1.3 Polymorphism & Algebraic Data Types

Smart supports polymorphic types and predicates. For declaring polymorphic types a number of type parameters must be introduced in the type's declaration. For example, an implicit type of polymorphic pairs can be declared as follows:

```
type pair<A, B>
```

This type is parameterized by two types `A` and `B`, which are the types of the first and second projection of the pair. Type variables must always start with an uppercase letter, while regular types must always start with a lowercase letter. The declaration of polymorphic predicates is straightforward. For instance, declaring an implicit constructor for the `pair` type declared above, amounts to the following:

```
public new_pair(A a, B b, pair<A, B> p+)
implicit program
```

This predicate is implicitly parameterized by two type variables, `A` and `B`. The type parameters of a predicate are implicitly determined by the type variables in its arguments. Local variables in explicit predicates can also be declared with polymorphic types. However, they can only depend on type variables introduced in the predicate's parameters. Type variables in polymorphic types can be instantiated by any type.

As mentioned in Section 3.1.1, Smart allows users to define their own concrete data types, by using algebraic data types, namely *structures* and *variants*.

Structures. Structures, also called records or tuples in other programming languages, represent the *Cartesian products* of the different types of their elements, called *fields*. In Smart, these can be declared in two manners: either by using the keyword **struct**, followed by the name of the structured type and its list of field types and field names, or, by using the keyword **type**, as shown below. The latter is preferred. Declaring polymorphic structures is possible by introducing type variables in the definition:

```
struct pair<A, B> {
  A fst;
  B snd;
}
```

```
type pair<A, B> = {
  A fst;
  B snd;
}
```

In order to build and manipulate structures, Smart supports built-in constructors and accessors. For instance, for the following type definition of a structure:

```
type t = {
  t1 f1;
  t2 f2;
  ...
  tn fn;
}
```

a constructor, a destructor, as well as individual accessors and "updaters" for any of the structure's fields are generated by Smart. Constructing an object of type `t` amounts to using `t@new` which requires a value for each of `t`'s fields. For example, creating a structure value `s` of type `t` with values `e1 ... en` for each field, amounts to calling `t@new(s+, e1, ..., en)`. The values of these fields can all be read with a single predicate call to `t@all(s, e1+, ..., en+)` (which "destructs" the structure value into its fields components). Individual accessors of type `t@fi(s, ei+)` are provided as well for any field `fi`. Finally, the value of a field `fi` can be set to some variable `vi` by using `t@fi(s+, vi)`. As all statements in Smart, this call has a functional nature and handles immutable data. Thus, setting the value of the `fi` field amounts to returning a *new* structure where all fields have the same value as `s`, except `fi` which is set to `vi`.

It is possible to define a structured type with no fields at all:

```
struct unit { }
```

The value `s` of this type can be constructed by using `unit@new(s+)` without any input. This type can be seen as representing the absence of information.

Variants. Many programs need to deal with heterogeneous collections of values. For example, a node in a binary tree can be either a leaf or an interior node with two children; similarly, a node of an abstract syntax tree in a compiler can represent a variable, an abstraction, an application, etc. Variant types provide the mechanism that supports this kind of heterogeneous value collections (Pierce, 2002).

Variants, also called tagged unions in other programming languages, can be seen as the dual of structures. A *variant* is the *disjoint union* of different types. It represents data that may take on multiple forms, where each form is marked by a specific tag called the *constructor*.

Revisiting our previously declared types `cartesian_point` and `polar_point`, in Smart we can define a type `point` as being either expressed in Cartesian, or in polar or spherical coordinates using the following variant declaration:

```
type point =
| Cartesian(cartesian_point p)
| Polar(polar_point p)
| Spherical(float r, float theta, float phi)
;
```

Each form that a variant can take is indicated by the symbol `|`, followed by the uppercase *tag* and the list of parameters and their types. The cases are mutually exclusive and a value of type `point` can have only one form at a time. An object of type `point` can be built by using one of the *constructors* called with the appropriate number and types of inputs. For instance, a Cartesian point `pc` can be obtained by calling: `point@Cartesian(p, pc+)`. Given an object `pt` of type `point`, we can also distinguish between the different cases by using a constructor that is similar to the `match ...with` construct in OCaml:

```
switch(pt)
case Cartesian (cartesian_point p) : get_X(p, x+);
case Polar (polar_point p) : get_radius(p, r+);
case Spherical (float r, float theta, float phi) : ...
```

For verifying if a given point `pt` is a `Cartesian` point, we can use:

```
point@case[Cartesian](pt)
```

This could be obtained using the `switch` construct, but for practical considerations the **case** construct has been additionally provided as a built-in predicate.

3.1.4 Specifications

Smart also supports various types of logical specifications, ranging from *axioms* and lemmas, to pre- and postconditions, invariants and inductives.

In Section 3.1.1, we stated that implicit predicates are a form of assumption and that declaring implicit Smart types and the predicates manipulating them, provides a convenient manner of axiomatizing external implementations, frequently developed in a lower-level language. They can provide implementation-independent descriptions and act as abstractions that hide hardware-related details and low-level implementation decisions. Another form of assumptions are *hypotheses*. Hypotheses are logical results that are assumed, i.e. they constitute *axioms* which are supposed to be true. In Smart, hypotheses are *specification-only* predicates, i.e. they cannot be called in the code. They are introduced by the keyword `hypothesis`.

For example, we could revisit our polymorphic pair type introduced in Section 3.1.3 and provide a polymorphic axiomatization for it by using implicit predicates and hypotheses, that stipulate that the operations `fst` and `snd`, retrieve the first and second, respectively, elements of the pair. These are declared as follows:

```
type pair <A, B>

public new_pair(A a, B b, pair<A, B> p+)
implicit program

public fst(pair<A, B> p, A a+)
implicit program

public snd(pair<A, B> p, B b+)
implicit program

public hypothesis pair_fst(A a, B b)
program {{ pair<A, B> p, A a2 }}
{
  new_pair(a, b, p+);
  fst(p, a2+);
  a = a2;
}
public hypothesis pair_snd(A a, B b)
program {{ pair<A, B> p, B b2 }}
{
  new_pair(a, b, p+);
  snd(p, b2+);
  b = b2;
}
```

Lemmas are another type of specification-only predicates, meant to facilitate proving logical properties. In contrast to hypotheses, lemmas must be proven. A lemma can be introduced with the keyword `lemma` and it states that all paths that exit from its body with an undeclared exit label represent impossible execution scenarios.

In Section 3.1.1, we introduced a type `cartesian_point` allowing to express a point by its Cartesian coordinates, and we defined a predicate `translate_point` for translating a point by a given pair of numerical values (a, b). We revisit our example and implement a predicate that translates a pair of points by a fixed pair of numbers (a, b) that are added to the Cartesian coordinates of each point of the pair. In addition, we consider an implicit predicate `euclidean_dist` that computes the Euclidean distance d:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

between a pair of points $\langle (x_1, y_1); (x_2, y_2) \rangle$. These are declared as follows:

```
type point_pair = pair<cartesian_point, cartesian_point>

/* For a pair of points ((x1, y1); (x2, y2)) compute
   d = sqrt((x2 - x1)^2 + (y2 - y1)^2) */
public euclidean_dist(point_pair p, float d+)
-> [true]
implicit program

/* For a pair of points ((x1, y1); (x2, y2)) and a fixed
   numerical pair (a, b), compute ((x1', y1'), (x2', y2'))
   as ((x1 + a, y1 + b), (x2 + a, y1 + b)). */
public translate_pair(point_pair p, pair<int, int> t,
                      point_pair o+)
-> [true]
{
...
}
```

The translation of a pair of points preserves the Euclidean distance between them: the Euclidean distance of a pair of points `p` will be equal to the Euclidean distance of the pair of points obtained after a translation. We can express this property by declaring it as a lemma:

```
public lemma edist_preserved(pair<float, float> t,
                             point_pair p)
program {{ point_pair translated, float d1, float d2 }}
 {
  euclidean_dist(p, d1+) =>
  translate_pair(p, t, translated+) =>
  euclidean_dist(translated, d2+) => d1 = d2;
 }
```

Specifying contracts for Smart predicates is also possible by employing pre- and postconditions. A *precondition* represents a logical property that must be true prior to calling a predicate and it serves the purpose of letting the callers know when it is safe to call some predicate. Typically, it represents the caller's obligations. In Smart, a precondition can be introduced with the keyword **pre** and it can be attached to any implicit or explicit predicate. A precondition can refer to the predicate's inputs and it can declare its own local variables. However, it cannot make use of the predicate's outputs.

For instance, for the `atan2` predicate discussed in Section 3.1.2, we could indicate that the predicate should never be called with the coordinates (0, 0) of the origin by adding the following precondition:

```
public const float ZERO

public atan2(float x, float y, float at+) -> [true]
pre {
 x != ZERO || y != ZERO;
}
implicit program
```

A *postcondition* represents a logical condition that must be true after executing a predicate. Its purpose is to indicate to the callers of a predicate what they are entitled to expect with respect to the outputs produced by the predicate. In Smart, postconditions are introduced with the keyword **post** and they can be attached to any implicit or explicit (computational) predicate, on a subset or all of the predicate's output labels. They can refer to the predicate's inputs and the outputs associated to the label considered in the postcondition. Additionally, they can declare their own local variables.

For instance, a predicate `equal_points` verifying if two points are equal and having four possible exit labels, `eq_points`, `eq_x`, `eq_y` and **false**, respectively, could declare postconditions as follows:

```
public equal_points (cartesian_point p, cartesian_point q)
-> [eq_points, eq_x, eq_y, false]
program {{ float px, float qx, float py, float qy }} {
  cartesian_point@x(p, px+);
  cartesian_point@x(q, qx+);
  cartesian_point@y(p, py+);
  cartesian_point@y(q, qy+);
  if px = qx;
  then {
    [true: eq_points, false : eq_x] py = qy; }
  else {
    [true: eq_y] py = qy; }
 }
post eq_points { p = q;}
```

```
post eq_x {{float x1, float x2}} {
    cartesian_point@equals[x](p,q);
 }
post * {p != q;}
```

The first postcondition applies to the exit label `eq_points`, the second to the label `eq_x` and the last one, indicated by `*` applies to labels `eq_y` and **false**.

In Smart, mathematical relations can be represented by introducing *inductives* or *schemes*. These predicates have no outputs but they always have **true** and **false** as their exit labels. *Inductive* predicates are the only part of the language that cannot be transformed into executable code; however, they can be used to facilitate the proofs. Predicates introduced with the **inductive** keyword represent the least fixed point of their *cases*, introduced with the keyword **case** and a user-defined name. Each case can introduce existentially quantified variables. In particular, in the absence of recursion, inductive predicates represent a *parallel disjunction* of cases. An inductive predicate will exit with the label **true** if any of its declared cases holds.

For example, we could specify membership for an implicit array type using an inductive named `contains` having a single case, with the user-defined name `ElemAt`, which introduces an existentially quantified variable `idx`:

```
type array<A>

public get_size(array<A> arr, int s+)
implicit program
public get_elem(array<A> arr, int i, A ai+)
-> [true, oob]
implicit program

// Membership defined with an inductive and an existential.
public contains(array<A> arr, A a) -> [true, false]
inductive
/* An array contains an element if there exists a valid
index where this element is to be found. */
case ElemAt(int idx): {{ A b }} {
  [ oob : false ] get_elem(arr, idx, b+) && b = a; }
```

Schemes on the other hand represent *conjunction* of cases; cases are introduced with the keyword `with`, followed by a user-defined name and each of them can introduce universally quantified variables. A scheme will return the label **true** only if all of its declared cases hold.

Using a scheme with two cases, `Size` and `Forall`, as shown below, we can define the pointwise equality of arrays. The first case, `Size`, verifies if the two arrays have the same length, by introducing two universally quantified variables, `n` and `m`. The `Forall` case verifies that for any index `i`, the arrays contain equal elements. Two arrays are

equal pointwise if and only if they are of the same size and at any given index `i`, the arrays have the same element.

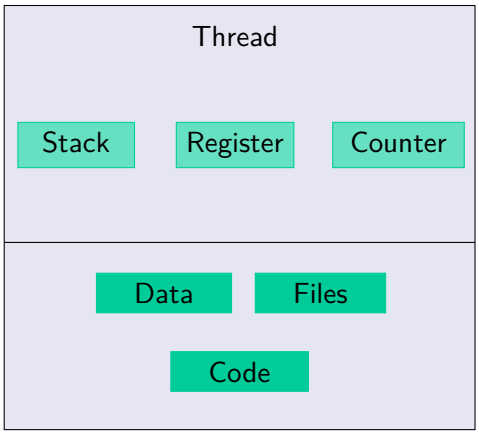
```
public equals_pointwise(array<A> arr1, array<A> arr2)
-> [true, false]
// Extensional equality of arrays [arr1] and [arr2]
scheme
// They must be of the same size
with Size: {{ int n, int m }} {
    get_size(arr1, n+) => get_size(arr2, m+) => n = m; }


// If they exist, elements at the same index must be equal
with Forall(int i): {{ A a, A b }} {
    ?get_elem(arr1, i, a+) => ?get_elem(arr2, i, b+) =>
    a = b; }
```

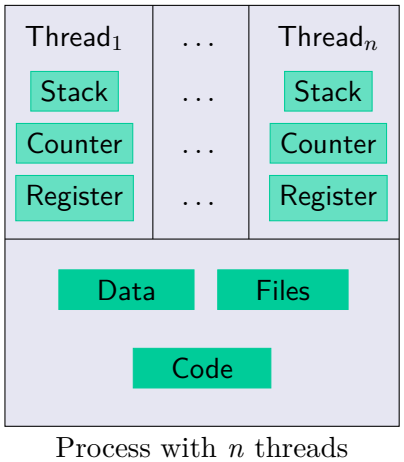
Loop invariants are supported as well. These can be introduced in various ways, for instance by declaring them with the keyword `invariant` or by declaring them as *inductives*.

3.1.5 Illustrating Smart – An Abstract Process Manager

To illustrate the Smart language and its capabilities, we consider an abstract process manager and its fundamental components, *process* and *thread*. We define the data structures corresponding to threads and processes, implement the predicates corresponding to a simple *thread switch* and specify some fundamental properties for processes.



Process with a single thread



Process with n threads

The implementation of threads and processes differs depending on the operating system, but frequently a thread is a component of a process that belongs to exactly one process, outside which it cannot exist. Each thread represents a separate flow of

control. Multiple threads can be associated to one process; they execute concurrently and provide a mechanism to improve application performance through parallelism. In a nutshell, threads represent a software approach to improving the performance of operating systems by reducing the overhead of process switching.

A thread is a flow of execution through the process code, having its own program counter, that keeps track of which instruction to execute next, as well as, system registers which hold its current working variables, and a stack which contains the execution history. Every thread is uniquely identified by a *thread identifier*. Peer threads share some information, such as the code and data segments. When one thread alters a code memory item, all other threads see the change.

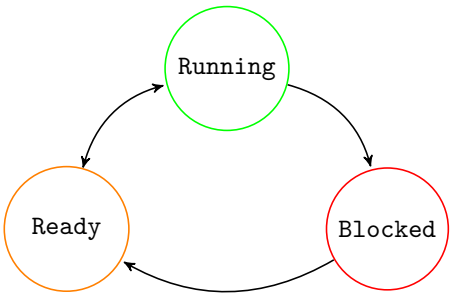


FIGURE 3.1 – Possible Transitions between Thread States

We define a thread type as a structure consisting of multiple fields such as the thread's identifier, its current state and the memory region for its stack.

```
type memory_region = {
 // Start address
 int start;
 // Region length
 int length;
}
```

```
type state =
  | Ready
  | Running
  | Blocked
;
```

```
type thread = {
 // Identifier
 int id;
 // Current state
 state crt_state;
 // Stack
 memory_region stack;
}
```

The thread's stack is identified by its start address and its length. The state of a thread is defined as a variant having three alternatives: Running (the thread is currently executing), Ready (the thread is currently awaiting execution and could potentially be started) and Blocked (the thread has exhausted its allocated time or is waiting for an event to occur; it must be unblocked before being able to execute). The possible transitions between states are shown in Figure 3.1. A thread's current state determines the valid transitions.

Similarly, a process is defined as a structure consisting of an internal identifier, an identifier for the thread that is currently executing, an address space and an array of possibly inactive threads associated with it. Whether a thread in the thread array is active or has terminated is indicated by a variant of type option. An *inactive* thread,

indicated by `None`, is a thread that terminated its execution and whose slot in the array of associated threads has not been reallocated. In contrast, a *blocked* thread, indicated by `Some`, is a thread that cannot execute currently but should execute in the future, once the resources it is waiting for are freed. We consider a *segmented address space*, with addresses existing not in a single linear range, but instead in multiple segments, corresponding to the code, the data and the stack, respectively.

```
type option<A> =
  | None
  | Some (A a)
;

type process = {
 // Array of associated threads
 array<option<thread>> threads;
 // Internal id
 int pid;
 // Currently running thread
 int crt_thread;
 // Address space
 address_space adr_space;
}
```

```
type address_space = {
 memory_region code;
 memory_region data;
 memory_region stack;
}
```

Next, we consider a simple predicate called `stop_thread`, having two possible execution scenarios as indicated by its two exit labels **true** and `invalid`. When the given input index `i` corresponds to an active thread, the predicate executes successfully, thus exiting with **true**. In this case, the state of the `i`-th thread associated to the input process is set to Blocked and the new state of the process is returned in the output `out`. Otherwise, when the given index `i` corresponds to a thread that is Ready, or when there is no active thread at that index, the predicate exits with the label `invalid` and no output is generated.

```
public stop_thread(process in, int i, process out+)
-> [true, invalid]
program {{array<option<thread>> ta, state s, thread ti,
          option <thread> tio}}
{
  // Copy in to out
  out := in;
  // Fetch in.threads and copy it to ta
  process@threads(in, ta+);
  // Get the array's i-th element
  [ oob : invalid ] get_elem(ta, i, tio+);
  // Check if the i-th element is active
  switch (tio)
   case Some (thread th): {ti := th;}
   case None: raise invalid;
```

```
  // Get the thread's current state
  thread@crt_state(ti, s+);
  //Check whether the transition is valid
  [false : invalid]state@case[Running](s);
  // Create the new state for the running thread
  state@Blocked(s+);
  // Set the newly created state
  thread@crt_state(ti+, s);
  // Reset tio to the thread with the modified state
  option@Some(tio+, ti);
  // Reset the i-th thread and return the new state ta
  [ oob : invalid ] set_ei(ta, i, tio, ta+);
  // Update out.threads to ta
  process@threads(out+, ta);
}
```

Another auxiliary predicate called `start_thread`, when given a valid index, of an unblocked thread, sets the state of the `i`-th thread to `Running`. It is implemented similarly as shown below:

```
public start_thread(process in, int i, process out+)
-> [true, invalid]
program {{array<option<thread>> ta, state s, thread ti,
          option <thread> tio}}
{
  // Copy in to out
  out := in;
  // Fetch in.threads and copy it to ta
  process@threads(in, ta+);
  // Get the array's i-th element
  [ oob : invalid ] get_ei(ta, i, tio+);
  // Check if the i-th thread is active
  switch(tio)
   case Some (thread th): {ti := th;}
   case None: raise invalid;
  thread@crt_state(ti, s+);
  //Check whether the transition is valid
  [false : invalid]state@case[Ready](s);
  // Create the new state for the running thread
  state@Running(s+);
  // Set the newly created state
  thread@crt_state (ti+, s);
  // Reset tio to the thread with the modified state
  option@Some(tio+, ti);
  // Set the i-th element and return the new state ta
  [ oob : invalid ] set_ei(ta, i, tio, ta+);
```

```
    // Update out.threads to ta
    process@threads( out+, ta);
  }
```

These two predicates will be called by the predicate `run_thread` that performs a simple thread switch. It stops the thread currently executing, indicated by `crt_thread` and starts the one with the given index `i`. The new state of the process is returned in the output `out`.

```
public run_thread(process in, int i, process out+)
-> [true, inval]
program {{ int crt }} {
  process@crt_thread(in, crt+);
  [true : true, invalid : inval] stop_thread(in, crt, out+);
  [true : true, invalid : inval] start_thread(out, i, out+);
  process@crt_thread(out+, nid);
}
```

Next, we introduce a fundamental property for any valid process state, namely the fact that the stack regions of all its associated threads are completely disjoint.

```
public not_disjoint(process p) -> [true, false]
inductive
case StacksJoint (int i, int j):
{{thread ti, thread tj, memory_region sti,
  memory_region stj}} {
  i != j;
  [None : false] thread(p, i, ti+);
  [None : false] thread(p, j, tj+);
  thread@stack(ti, sti+); thread@stack(tj, stj+);
  overlap(sti, stj);
}
case CodeStackJoint (int i):
 {{thread ti, memory_region sti, address_space as,
   memory_region code}} {
   [None : false] thread(p, i, ti+);
   thread@stack(ti, sti+);
   process@adr_space(p, as+);
   address_space@code(as, code+);
   overlap(sti, code);
}
case DataStackJoint (int i):
 {{thread ti, memory_region sti, address_space as,
   memory_region data}} {
   [None : false] thread(p, i, ti+);
   thread@stack(ti, sti+);
```

```
    process@adr_space(p, as+);
    address_space@data(as, data+);
    overlap(sti, data);
  }
  public disjoint_stacks(process p) -> [true, false]
  program {
    !not_disjoint(p);
  }
```

This property is expressed using an inductive predicate that characterizes the potential situations in which the memory isolation of the different associated threads of a process can be broken. The natural manner of expressing such a property in Smart is by using a scheme as presented in Section 3.1.4; here we use an inductive predicate because the language we are working with and which will be presented in Chapter 4 does not support schemes. In our inductive predicate, the first case, `StacksJoint`, checks whether there exist two different threads having *overlapping* stacks. The next two cases, `CodeStackJoint` and `DataStackJoint`, check whether there exists a thread whose stack overlaps the process' code segment or data segment, respectively. This uses an auxiliary predicate, verifying if two memory regions overlap, i.e. if there exists an address that is contained simultaneously by two different segments. This operation is symmetric; we express this property with the lemma `overlap_sym`.

```
public contains (memory_region m, int address)
-> [true, false]
implicit program

public overlap(memory_region m1, memory_region m2)
-> [true, false]
inductive
case InBoth (int address): {
  contains(m1, address) && contains(m2, address);
}

public lemma overlap_sym(memory_region m1, memory_region m2)
-> [true, false]
program {
  overlap(m1, m2) => overlap(m2, m1);
}
```

3.2 ProvenTools

ProvenTools is a comprehensive set of development tools for the Smart language. It has been developed at Prove & Run with the goal of facilitating the achievement of high-level certifications. The toolchain has the structure of a set of Eclipse plug-ins of JDT type – Java Development Tools. Together, these constitute a complete Integrated Development Environment (IDE) allowing one to not only write, edit and document

Smart models, but also to browse proof obligations, to prove them by employing a built-in prover and finally, to generate executable code in C or Java.

The plug-ins are based on Xtext (*Xtext Documentation*), an official Eclipse plug-in dedicated to the creation of DSLs (Domain Specific Languages) in Eclipse. Xtext-based DSLs are described in an EBNF (Extended Backus-Naur Form) grammar language. Fully statically typed expressions can be embedded in the developed DSL and Java style scoping and linking are supported.

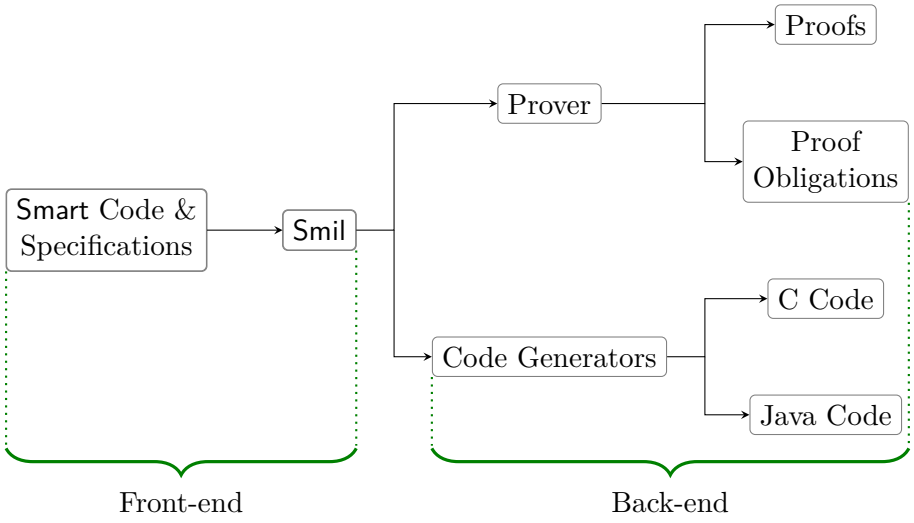


FIGURE 3.2 – The ProvenTools Toolchain

Concretely, the toolchain includes a compiler whose front-end contains the plug-in in charge of Smart, as well as the plug-in dedicated to Smil, the *Smart Intermediate Language*, to which Smart programs and specifications are translated. Smil is a simpler form of the Smart language. Though roughly equivalent to Smart, Smil has a rather different form manipulating less complex structures and having no syntactic sugar. Harder to be understood by a human reader, Smil is meant to be easily manipulated by the back-end of the toolchain. The back-end currently offers a C code generator and an interactive prover. An overview of this architecture is shown in Figure 3.2.

While employing ProvenTools, the code undergoes various compilation steps and transformations. During the compilation chain, the Smart code is transformed to a Smart AST (Abstract Syntax Tree). The obtained AST is then compiled to a Smil AST. Following, the Smil AST is transformed to Smil source code and then reinserted in the compilation chain by the plug-in in charge of it.

After finishing all the compilation chain and obtaining the Smil AST and the associated Smil source code, the back-end of the compiler can be employed. The back-end comprises a source code generator and a prover. The generator transforms Smart models into their equivalents in C.

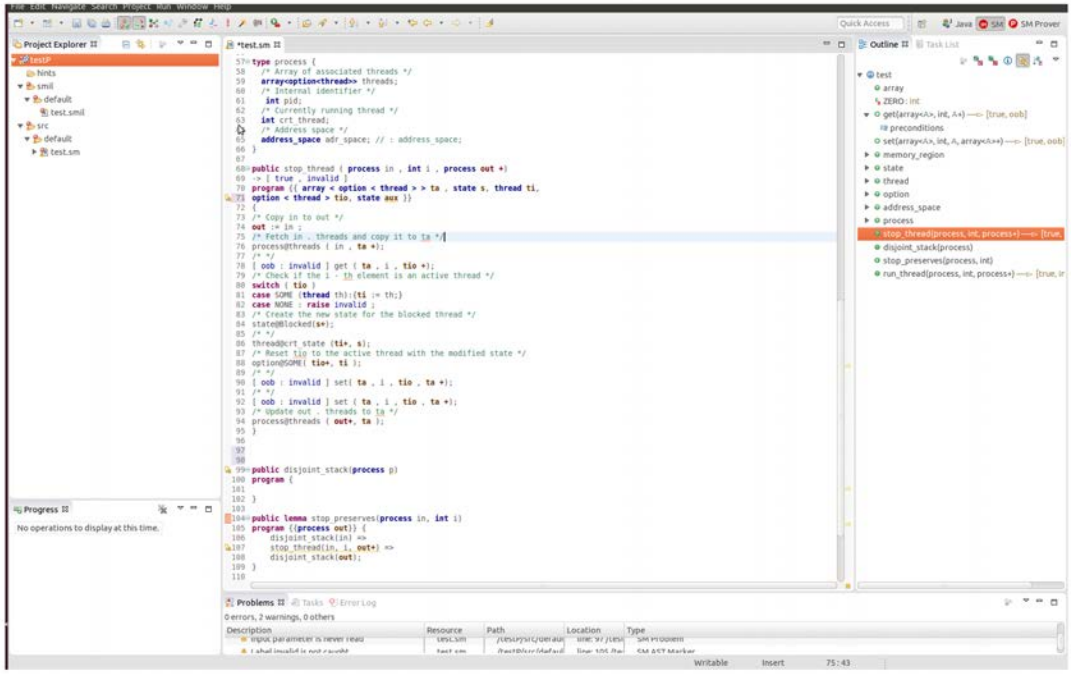


FIGURE 3.3 – Smart Editor

Smart Editor. The Smart editor provides facilities to edit Smart code and supports broad and complex features, such as syntax highlighting, facilities for code navigation and visualization, and edition assistants, including word completion and quick fixes. A snapshot of it is shown in Figure 3.3.

Prover. ProvenTools provides users a dedicated view for interacting with the prover. This presents the existing proof obligations and provides facilities to solve them. Proof obligations are generated for any logical lemma, precondition, postcondition or invariant included in the Smart models. Additionally, any label that remains unhandled in the code triggers the generation of a proof obligation, thus enforcing that each possible exit label of a predicate is either explicitly handled or proven to be impossible.

An automatic prover, trying various proof search procedures, is called whenever a proof obligation is generated. It uses previously proven obligations or existing hypotheses for discharging new obligations automatically. Unproved obligations can be solved by interactively employing manual tactics, called *hints*, which are provided in the IDE. Hints that are considered useless with respect to the currently selected proof obligations are automatically disabled. Additionally, users can define *strategies*, i.e. proof patterns, and employ an interactive proof assistant, that applies them automatically in the background. This will suggest a possible proof as soon as it finds one. Proofs thus found are rechecked as if they had been done manually.

ProvenTools offers facilities to inspect any manual or automatic proof step, thus making an eventual review of the proofs possible. The toolchain also provides a dedicated system for assisting the user into adapting former proofs to new changes, due to code maintenance or evolution.

C Code Generator. The executable part of Smart models is translated to executable C code by the C code generator. To this end, the executable parts of the Smart models are identified and extracted, while the logical parts are discarded. Users can guide this process through annotations and they can specify that particular values are purely logical. Functional implementations are transformed to imperative ones: the dedicated C code generation plug-in tries to replace functional modifications of structures in the models by in-place updates. Such transformations are correct only if the different values are handled linearly in the Smart code, i.e. if no previous value is read after applying a functional update on it. For ensuring the safety of functional to imperative code transformations, the C generation plug-in employs various global static analyses. When safety cannot be guaranteed, the generator reports errors or introduces copies, if the users deemed it acceptable.

In earlier experiments (Lescuyer, 2015), the Prove & Run team was able to generate C code for a complete model of ProvenCore that did not require dynamic allocation, and ran at a speed comparable to the original C code.

3.3 Smil

Smil is an intermediate language to which Smart models are compiled. Similarly to Smart, Smil is a functional language with algebraic data types (structures and variants). However, unlike Smart, Smil is not a user-oriented language, i.e. it was not designed to write programs in it directly, but rather to provide a representation of Smart programs at a different level of abstraction. Thus, reading Smil code is a rather cumbersome task as it is a language without syntactic sugar, meant to serve as a starting point for the main components of the ProvenTools back-end exploiting Smart models: the prover and the code generator.

To give an idea of Smil's syntax, we illustrate below the types `thread` and `process`, as well as the `stop_thread` predicate, from our abstract process manager example given in Section 3.1.5.

```
public type state =
  | Ready
  | Running
  | Blocked;
```

```
public type thread = {
  id : int;
  crt_state : state;
  stack : memory_region;
}
```

```
public state_acopy_ahypothesis(state state_1) -> [true]
hypothesis
{{state state_2}}{
 [<1>] : state@switch(state_1)
         -> [Ready -> 5, Running -> 4, Blocked -> 3];
 [<2>] : ==<state>(state_1, state_2)
         -> [true -> true, false -> error];
 [<3>] : state@Blocked(state_2)
         -> [true -> 2];
 [<4>] : state@Running(state_2)
         -> [true -> 2];
 [<5>] : state@Ready(state_2)
         -> [true -> 2];
}

public thread_ahypothesis(thread x1) -> [true]
hypothesis
{{thread x2, int zid, state zcrt_state,
  memory_region zstack}}{
 [<1>] : thread@all(x1, zid, zcrt_state, zstack)
         -> [true -> 2];
 [<2>] : thread@new(x2, zid, zcrt_state, zstack)
         -> [true -> 3];
 [<3>] : ==<thread>(x1, x2)
         -> [true -> true, false -> error];
}
```

The type declarations in Smil strongly resemble their Smart counterpart. Predicate declarations as well mirror the form found in Smart, except that in Smil any output variable associated to the **true** exit label is explicitly declared as such. Preconditions and postconditions are appended to any predicate and, as shown above, a hypothesis is added for any explicitly declared type.

The real syntax differences are visible in predicate implementations: every statement is preceded by a numerical label and every possible exit label `lbl` of the statement indicates another numerical label. The latter numerical label actually designates the statement that will be executed next, if the current statement exits with label `lbl`. In particular, this mechanism replaces the **try** ...**catch** ... and the conditional control constructs, as well as the logical operators and any other construct based on label transformers described in Section 3.1.2. Thus, the predicate bodies are very similar in form to a control flow graph, where the statements represent the nodes of the graph and the exit labels represent transitions.

```
public stop_thread(process in, int i, process out+)
 -> [true<out>, invalid]
pre { [<0>] : true() -> [true -> true]; }
```

```
{{ array<option<thread>> ta, state s, thread ti,
   option<thread> tio, thread th}} {
[<1>] : :=<process>(out, in)
        -> [true -> 2];
[<2>] : process@threads(in, ta)
        -> [true -> 3];
[<3>] : get<option<thread>>(ta, i, tio)
        -> [true -> 4, oob -> invalid];
[<4>] : option@switch<thread>(tio, th)
        -> [None -> 6, Some -> 7];}
[<5>] : state@Blocked(s)
        -> [true -> 8];
[<6>] : true()
        -> [true -> invalid];
[<7>] : :=<thread>(ti, th)
        -> [true -> 5];
[<8>] : thread@crt_state+(ti, ti, s)
        -> [true -> 9];
[<9>] : option@Some<thread>(tio, ti)
        -> [true -> 10];
[<10>] : set<option<thread>>(ta, i, tio, ta)
        -> [true -> 11, oob -> invalid];
[<11>] : set<option<thread>>(ta, i, tio, ta)
        -> [true -> 12, oob -> invalid];
[<12>] : process@threads+(out, out, ta)
        -> [true -> true];
}
post true 0
post invalid 0
```

In a nutshell, Smil constitutes a representative, albeit restricted set of constructs and it is a language designed to be well-suited for further transformations and analyses.

The next chapter focuses entirely on αSmil, the computational version of Smil with which we are working throughout the rest of this thesis. We will illustrate its usage and describe its abstract syntax and formal semantics.

Chapter 4

The αSmil Language

> One day I will find the right words,
> and they will be simple.
>
> Jack Kerouac

In this chapter, we define the syntax and the semantics of αSmil, the language that we consider in this thesis. This is a computational version of Smil (presented in Section 3.3) which is essentially a subset of Smart, presented in the previous chapter, Chapter 3. However, it contains a few additional elements introduced for the purpose of this thesis.

The αSmil language is a first-order, purely functional and strongly-typed language with arrays and algebraic data types, i.e. structures and variants. It is an intermediate, analysis-oriented language.

4.1 αSmil Syntax

The αSmil language is minimal in the sense that it contains only those constructs that are needed for the purpose of this thesis. For instance, unlike Smart and Smil, the language does not contain *visibility modifiers* because these modifiers play no role in the techniques presented in the sequel. During the introduction of the grammar, we will point out the most important deviations from Smart and Smil.

Programs. A program in αSmil consists of a number of type and constant declarations and definitions followed by a collection of predicates. In contrast to Smart and Smil, type and predicate declarations have no visibility modifiers (such as **public**), and they are not organized into *modules*. The absence of visibility modifiers is a natural consequence of the disappearance of modules. We assume that there is one module in which every type, constant and predicate declaration resides and these are mutually visible to each other. These restrictions are made for the sake of simplicity since the techniques proposed in this thesis are orthogonal to the concepts of visibility and modules.

Constants are declared using the keyword **const**, followed by the type and the constant identifier. Constant identifiers are written in upper-case letters and are preceded by the special symbol `#`.

Types are declared using the keyword **type**, followed by the type identifier and, optionally, in the case of polymorphic type declarations, by a number of type parameters, given in upper-case letters between <>. In the case of implicit types this constitutes the complete type declaration. Explicit type declarations continue with the symbol = and the type's definition. Throughout the rest of this chapter and the presentation of our static analyses we will ignore polymorphism. The abstract types of our analyses are not polymorphic and the impact of polymorphism is visible only at the implementation level, for type substitutions that will be discussed in Chapter 8.

Types. Similarly to Smart, algebraic data types, i.e. structures and variants, and associative arrays are supported. We let $\mathbb{T}$ be the universe of type identifiers and $T_0 \subset \mathbb{T}$ the set of base type identifiers. We assume a set of identifiers for structure fields and variant constructors, denoted by $\mathcal{F}$ and $\mathcal{C}$, respectively.

A *structure* represents the *Cartesian product* of the different types of its elements, called *fields*. A *variant* is the *disjoint union* of different types. It represents data that may take on multiple forms, where each form is marked by a specific tag called the *constructor*. *Arrays* group elements of data of the same type (given in angle brackets) into a single entity; elements are selected by an index whose type is included (as denoted by the superscript) in the array's definition as well.

Definition 4.1.1. Types $\tau \in \mathbb{T}$ in αSmil.

$$
\begin{array}{lllll}
\tau \in \mathbb{T}, \tau := & \mid \tau_0 \in T_0 & & & \text{base types} \\
& \mid \boldsymbol{struct}\{f_1 : \tau, \ldots, f_n : \tau\} & f_i \in \mathcal{F}, 0 \leq n & & \text{structures} \\
& \mid \boldsymbol{variant}[C_1 : \tau \mid \ \ldots \mid C_n : \tau] & C_i \in \mathcal{C}, 1 \leq m & & \text{variants} \\
& \mid \boldsymbol{arr}^{\tau}\langle \tau \rangle & & & \text{arrays}
\end{array}
$$

Variants and structures can be used together to model traditional algebraic variants with zero or several parameters. For instance, a generic type `option<T>` is actually modeled as:

$$\boldsymbol{variant}[\texttt{Some} : \boldsymbol{struct}\{\texttt{t} : \texttt{T}\} \mid \texttt{None} : \boldsymbol{struct}\{\}].$$

Concretely, structures are declared and defined by indicating a set of pairs of field identifiers and their corresponding types between {}. Declaring structures with *no fields* is possible. Variants are declared and defined by indicating the list of their constructors, each starting with an upper-case letter, preceded by the symbol |. Unlike structures, variants must have at *least one* declared constructor. For instance, the `state` and `thread` types from our Abstract Process Manager example given in Smart in Section 3.1.5, on page 48, have the following Smil declaration:

```
type state =
|Ready
|Running
|Blocked
```

```
type thread = {
  id : int;
  crt_state : state;
  stack : memory_region
}
```

In contrast to Smart, in structure declarations, the field name precedes the field type.

Predicates. Predicates are declared using the keyword **predicate**, which is specific to αSmil, followed by a *predicate identifier* and a signature. A *signature* is given by a sequence of *input* types and a non-empty finite mapping of exit labels, $\lambda \in \mathcal{L} \setminus \{\mathsf{error}\}$, to sequences of *output* labels. The set of exit labels $\mathcal{L}$ contains three distinguished elements, true, false and error. The latter cannot appear in predicate signatures; it is used as a sink node in control flow graphs, which will be presented in Section 4.2. We write signatures in the following manner:

$$\sigma = \underbrace{(x_1 : \tau_1, \ldots, x_n : \tau_n)}_{\text{input identifiers : types}} \underbrace{[\lambda_1 : (\tau_{11} : y_{11}, \ldots, \tau_{1k_1} : y_{1k_1}) \mid \ldots \mid \overbrace{\lambda_p : (\tau_{p1} : y_{p1}, \ldots, \tau_{pk_p} : y_{pk_p})}^{\text{label : (output types : identifiers)}}]}_{p \text{ possible exit labels}}$$

We denote by Σ the mapping between predicate identifiers and their signatures.

The predicate declaration is followed by the predicate's body. Depending on its body's nature, a predicate will be *implicit*, *explicit* or *inductive*. Smart implicit and explicit predicates have been presented in Section 3.1.1 of our previous chapter, while inductive predicates have been illustrated in Section 3.1.4, on page 46. For implicit predicates, the body consists solely in the keyword **implicit**. For explicit predicates, an optional *declaration unit* can follow. This is a finite mapping from variables to types and it must be given between double curly braces, i.e. {{`typeid videntifier`}}. Input and output parameters must be different from all the variables appearing in the declaration units. Declaration units are followed by a sequence of statements representing calls to predicates.

Just as presented in Chapter 3.1.4 for Smart, an inductive predicate is syntactically distinguished by the keyword **inductive**, followed by its different cases, declared with the keyword **case** followed by an identifier, an optional list of existentially quantified variables and a body of statements.

A generic call to a predicate p is of the form:

$$p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m].$$

The predicate p is called with inputs $e_1, \ldots, e_n$ and yields one of the declared exit labels $\lambda_1, \ldots, \lambda_m$, each having its own set of associated output variables $\bar{o}_1, \ldots, \bar{o}_m$, respectively. We denote by $\bar{o}$ a sequence of 0 or more output variables.

Statements. The αSmil language supports the statements presented in Table 4.2. These represent calls to built-in predicates and can be seen as special cases of the predicate call presented above. All statements have a functional nature and handle immutable data. A statement consists in as many variables as there are input types

$s :=$	$\mid o := e$	(1)	assignment
	$\mid e_1 = e_2$	(2)	equality test
	$\mid \textbf{\textit{nop}}$	(3)	no operation
	$\mid r := \{e_1, \ldots, e_n\}$	(4)	create structure
	$\mid \{o_1, \ldots, o_n\} := r$	(5)	destructure structure
	$\mid o := r.f_i$	(6)	access field
	$\mid r' := \{r \textbf{\textit{ with }} f_i = e\}$	(7)	update field
	$\mid r' = \langle f_1, \ldots, f_k \rangle r''$	(8)	check (partial) structure equality
	$\mid v := C_p[e]$	(9)	create variant
	$\mid \textbf{\textit{switch}}(v) \textbf{\textit{ as }} [o_1 \mid \ldots \mid o_n]$	(10)	destructure variant
	$\mid v \in \{C_1, \ldots, C_k\}$	(11)	variant possible
	$\mid o := a[i]$	(12)	array access
	$\mid a' := [a \textbf{\textit{ with }} i = e]$	(13)	array update
	$\mid p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]$	(14)	predicate call

TABLE 4.2 – αSmil – Set of Supported Statements

in the signature σ_p of the called built-in predicate p, and a mapping associating to each exit label of σ_p a sequence of variables, one variable for each output type in the corresponding sequence.

The first three statements are generic and can be applied to any type. Statement (1) is a call to the built-in assignment predicate denoted by `:=`, present in an identical form in Smart as well. Statement (2) is a call to the logical operator `=` verifying whether its two input arguments are equal. Statement (3) is the αSmil equivalent of a *no-operation*. As a general convention for the statements notation, we denote by e the identifiers of *entry* variables, and by o, the identifiers of *output* variables.

Statements (4) – (8) are structure-related. The first of them, statement (4), is the constructor of a structure `r` of type rtype, having n fields. It corresponds to the statement `rtype@new(r+, e_1, ..., e_n)` in Smart. Statement (5) returns the values of all the fields of r into the output parameters $o_1, \ldots, o_n$ and it is the equivalent of `rtype@all(r, o_1+, ..., o_n+)` in Smart. Statement (6) is the individual accessor of a field f_i and corresponds to `rtype@f_i(r, e_i+)` in Smart. As previously mentioned, our language is purely functional and handles only immutable algebraic data structures and arrays. Therefore, setting the field f_i of a structure, shown in (7) and being the equivalent of `rtype@f_i(r'+, e_i)`, returns a new structure where all fields have the same value as in r, except f_i which is set to e_i. Statement (8) verifies if the values of the indicated subset of fields of two structures r' and r'' are equal. It exists in Smart as well, where it has a similar syntax: `rtype@equals[f,g](r', r'')`, for checking that the values of fields `f` and `g` of the two structures are equal, or the dual: `rtype@equals-[f,g](r', r'')`, for checking that the values of all fields except `f` and `g` are equal.

The next group of statements is variant-related. The first of them, statement (9) creates a new variant v of type vtype using the constructor C_p with e as an argument. It corresponds to `vtype@Cp(v+, e)` in Smart. Statement (10) is used for matching on

the different constructors of the input variant v and corresponds to `switch(v) case ...` in Smart. The last statement of this group, statement (11), verifies if the given variant was created with one of the constructors in $\{C_1, \ldots, C_k\}$. This could be obtained with a variant switch, but for practical considerations it has been provided as a built-in predicate. Its counterpart in Smart is `vtype@case[C1, ..., Ck](v)`.

Statements (12) and (13) are array-related. (12) returns the value of the i-th cell of the input array a. Similarly to (7), updating the i-th cell of an array – shown in (13) – has a functional nature. It returns a new array where all cells have the same values as in a, except the i-th cell which is set to e. These statements are specific to αSmil.

Statement (14) is a generic call to a predicate p and has been presented on page 61.

Exit Labels. All of the built-in supported statements have an associated set of exit labels, $\lambda \in \mathcal{L} \setminus \{\mathsf{error}\}$. These are indicated in Table 4.3. There are two distinguished exit labels, true and false, respectively. An additional built-in label called error is used as a sink node in control flow graphs. It cannot be used as an exit label for a predicate.

TABLE 4.3 – Statements and their Exit Labels

Statement		Exit Labels
$o := e$	(1)	$\left[\ \mathsf{true} \mapsto o\ \right]$
$e_1 = e_2$	(2)	$\left[\begin{array}{l}\mathsf{true} \mapsto \emptyset \\ \mathsf{false} \mapsto \emptyset\end{array}\right]$
nop	(3)	$\left[\ \mathsf{true} \mapsto \emptyset\ \right]$
$r := \{e_1, \ldots, e_n\}$	(4)	$\left[\ \mathsf{true} \mapsto r\ \right]$
$\{o_1, \ldots, o_n\} := r$	(5)	$\left[\ \mathsf{true} \mapsto o_1, \ldots, o_n\ \right]$
$o := r.f_i$	(6)	$\left[\ \mathsf{true} \mapsto o\ \right]$
$r' := \{r\ \textbf{\textit{with}}\ f_i = e\}$	(7)	$\left[\ \mathsf{true} \mapsto r'\ \right]$
$r' = \langle f_1, \ldots, f_k \rangle r''$	(8)	$\left[\begin{array}{l}\mathsf{true} \mapsto \emptyset \\ \mathsf{false} \mapsto \emptyset\end{array}\right]$
$v := C_p[e]$	(9)	$\left[\ \mathsf{true} \mapsto v\ \right]$

$\boldsymbol{switch}(v)\ \boldsymbol{as}\ [o_1 \mid \ldots \mid o_n]$	(10)	$\begin{bmatrix} \lambda_{C_1} & \mapsto & o_1 \\ \vdots & \ddots & \vdots \\ \lambda_{C_n} & \mapsto & o_n \end{bmatrix}$
$v \in \{C_1, \ldots, C_k\}$	(11)	$\begin{bmatrix} \mathsf{true} \mapsto \emptyset \\ \mathsf{false} \mapsto \emptyset \end{bmatrix}$
$o := a[i]$	(12)	$\begin{bmatrix} \mathsf{true} \mapsto o \\ \mathsf{false} \mapsto \emptyset \end{bmatrix}$
$a' := [a\ \boldsymbol{with}\ i = e]$	(13)	$\begin{bmatrix} \mathsf{true} \mapsto a' \\ \mathsf{false} \mapsto \emptyset \end{bmatrix}$
$p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]$	(14)	$\begin{bmatrix} \lambda_1 & \mapsto & \bar{o}_1 \\ \vdots & \ddots & \vdots \\ \lambda_m & \mapsto & \bar{o}_m \end{bmatrix}$

As shown in Table 4.3, statement (10) has an exit label λ_{C_i} corresponding to each constructor C_i of the input variant. Statements (2), (8) and (11) are bi-labeled, using true and false as logical values. Neither of them has any associated outputs. Statements (12) and (13) are bi-labeled as well. However, unlike the previously mentioned statements, they use the label false as an "out of bounds" exception and generate an output only for the label true. All other statements except (14) are uni-labeled: they associate all their output parameters (if any), to the label true. In contrast to Smart, in αSmil, every exit label, including true, must be explicitly indicated. Furthermore, any output is explicitly associated to an exit label.

In Section 3.1.5 (on page 50) of our previous chapter, we introduced a Smart predicate, called `stop_thread`. If the given index `i` designates an active associated thread, this predicate sets its state to `Blocked` and returns the new state of the process. Otherwise, the predicate exits with label `invalid`. Revisiting it, we can finally indicate its body in the αSmil language[1]:

TABLE 4.4 – Predicate Body in αSmil

```
// Signature
predicate stop_thread (process p, int i)
-> [true: process o | invalid]
// Declaration unit
{{array<option_thread> ta, option_thread th,
  thread ti, state s}}
// Predicate body
```

[1]The αSmil version is slightly simplified, as we are not checking if the transition to Blocked is valid.

```
{
  ta := p.threads : [true -> 1]; // 0
  th := ta[i] : [true -> 2, false -> 9]; // 1
  switch(th) as [ti | ] : [Some -> 3, None -> 9]; // 2
  s := Blocked  : [true -> 4]; // 3
  ti := {ti with crt_state = s} : [true -> 5]; // 4
  th := Some(ti) : [true -> 6]; // 5
  ta := [ta with i = th] : [true -> 7, false -> 9]; // 6
  o := {p with threads = ta} : [true -> 8]; // 7
  [true]; // 8
  [invalid] // 9
}
```

Every statement in our `stop_thread` example is followed by a construct of the form `exit_label -> numerical_label`. This indicates the statement to be executed next, as identified by the `numerical_label`, if the current statement exits with label `exit_label`. For example, when the first statement, `ta := p.threads`, exits with label **true**, the predicate's execution continues with the statement following it, having the numerical label `1`. We remark that the predicate's exit labels are included in the body of an explicit predicate, as can be seen at lines 8 and 9, respectively, in the case of **true** and `inval`. Intuitively, the predicate's body resembles a control flow graph and can be illustrated as shown in Figure 4.1. The predicate's exit labels are the control flow graph's exit nodes, as will be discussed in Section 4.2.

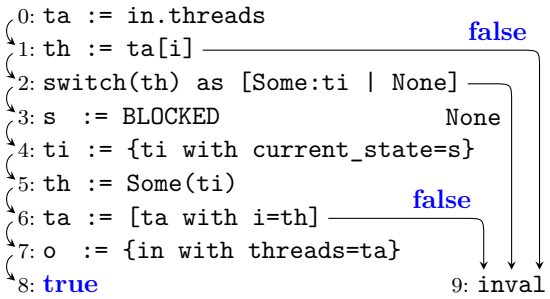


FIGURE 4.1 – Body of the `stop_thread` Predicate

We are working with αSmil, which is a computational version of Smil where all specification-only predicates have been removed. Simulating hypotheses, lemmas and contracts is straightforward and can be achieved using predicates having only the true and false labels and no associated output. Inductives are the only exception to this rule: they are supported in αSmil as well and their declaration is similar to the one in Smart. The αSmil equivalent of the `not_disjoint` inductive presented in our *Abstract Process Manager* example (on page 46) has the following form:

```
predicate not_disjoint(process p)
-> [true | false]
inductive
```

```
case StacksJoint (int i, int j) :
 {{thread ti, thread tj, memory_region sti,
  memory_region stj}} {
  i = j : [true -> 1, false -> 7];
  thread(p, i)[true: ti | None] : [true -> 2, None -> 7];
  thread(p, j)[true: tj | None] : [true -> 3, None -> 7];
  sti := ti.stack : [true -> 4];
  stj := tj.stack : [true -> 5];
  overlap(sti, stj)[true|false] : [true -> 6, false -> 7];
  [true];
  [false];
  [error]
 }
case CodeStackJoint (int k) :
 {{thread tk, memory_region stk, address_space asp,
   memory_region code}} {
  thread(p, k)[true: tk | None] : [true -> 1, None -> 6];
  stk := tk.stack : [true -> 2];
  asp := p.adr_space : [true -> 3];
  code := asp.code : [true -> 4];
  overlap(stk, code)[true|false] : [true -> 5, false -> 6];
  [true];
  [false];
  [error]
 }
case DataStackJoint (int l) :
 {{thread tl, memory_region stl, address_space aspace,
   memory_region data}} {
  thread(p, l)[true: tl | None] : [true -> 1, None -> 6];
  stl := tl.stack : [true -> 2];
  aspace := p.adr_space : [true -> 3];
  data := aspace.data : [true -> 4];
  overlap(stl, data)[true|false] : [true -> 5, false -> 6];
  [true];
  [false];
  [error]
}
predicate disjoint_stacks(process p) -> [true | false]
 {
  not_disjoint(p)[true|false] : [true -> 1, false -> 2];
  [true];
  [false];
  [error]
 }
```

This inductive predicate has been introduced and explained in Section 3.1.5 of the previous chapter (on page 52) and it characterizes the potential situations in which the memory isolation of the different associated threads of a process can be broken.

4.2 Control Flow Graph

Predicate bodies in αSmil resemble a *control flow graph* representation, having statements as nodes. The nodes represent program states, and the edges are defined by statements with a particular exit label λ.

The control flow graph $G_p = (N, \mathcal{E})$ of a predicate p has a node $n_i \in N$ for each program point. For each statement s at program point n_i that can execute and reach program point n_j with exit label λ_k, an edge (n_i, n_j) is added to G_p and labeled with s and λ_k. G_p has a single entry node $n_{in} \in N$ corresponding to the program point associated to the first statement of p. The set of exit nodes $n_{out} \subset N$ consists of the nodes associated to each possible exit label λ_k of the predicate. To these, one additional exit node which is used as a sink node is added. This corresponds to the error label.

In practice, all the outgoing edges of a node $n_i \in N$ bear the different cases of the same statement s found at program point n_i. Thus, the edges are labeled with the same statement s and there is an edge labeled s, λ_k for each possible exit label λ_k of s.

The subfigures in Figure 4.2 show the control flow graph of the following predicate:

```
predicate thread(process p, int i)
-> [true: thread ti | None | oob]
```

which receives a process `p` and an index `i` as inputs and returns the `i`-th active thread of the input process. If the `i`-th thread is inactive, it exits with the exit label `None`. In the case of an "out of bounds" exception, the exit label `oob` is returned. For better readability, Figure 4.2-b gives the control flow of the same predicate where we have labeled the nodes with statements of the predicate and the edges with their exit labels. Throughout the rest of our αSmil predicate examples, we will favour the latter representation.

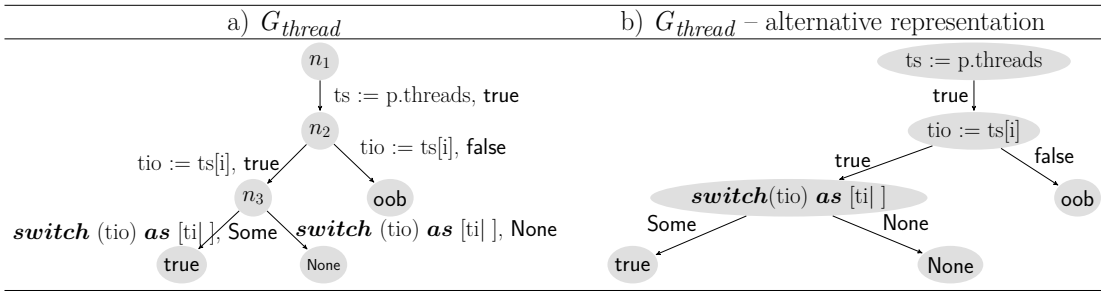


FIGURE 4.2 – Example – Control Flow Graph of Predicate `thread`

4.3 Well-Typed αSmil Statements

We formally define what it means for an αSmil statement to be well-typed and detail the full system of inference rules for the statements supported by αSmil in Table 4.6

and Table 4.7.

A well-typed αSmil statement is a statement that is compatible with the types specified in the signature σ_p of the called built-in predicate p. This requires a typing environment Γ mapping variables to their types.

Definition 4.3.1. Typing Environment Γ.

$$\Gamma : \mathcal{V} \to \mathbb{T}.$$

Furthermore, αSmil distinguishes between variables $v \in \mathcal{V}$ which can be written to and variables which are *read-only*. Therefore, the definition of well-typedness for statements requires two different sets of variable identifiers, one for each kind of variable. These are:

- $\mathcal{V}^+$, $\mathcal{V}^+ \subseteq \mathcal{V}$, which denotes the set of identifiers of writable and readable variables, and
- $\mathcal{V} \setminus \mathcal{V}^+$, which denotes the set of *read-only variables*.

The mapping between predicate identifiers and their signatures is denoted by Σ.

Definition 4.3.2. Mapping between Predicate Identifiers and Signatures.

$$\Sigma : \mathscr{P} \to \mathscr{S}.$$

Definition 4.3.3. Well-Typed Statement. A statement s exiting with label $\lambda \in \mathcal{L} \setminus \{\mathsf{error}\}$ is well-typed in the typing environment Γ, given Σ:

$$\Sigma, \Gamma, \mathcal{O} \vdash s \to \lambda$$

if it is compatible with the types specified in its signature. Moreover, outputs of a well-typed statement must be in the writable variables set, $\mathcal{O} \subseteq \mathcal{V}^+$.

The inference rule for a well-typed predicate call captures all these properties and is shown in rule [WTPCall], given in Table 4.6.

Table 4.6 – Well-Typed Predicate Call

$$\frac{\begin{array}{c}\Sigma(p) = (x_1 : \tau_1, \ldots, x_n : \tau_n)[\lambda_1 : (\tau_{1,1} : y_{1,1}, \ldots, \tau_{1,k_1} : y_{1,k_1}) \mid \ \ldots \\ \ldots \mid \lambda_m : (\tau_{m,1} : y_{m,1}, \ldots, \tau_{m,k_m} : y_{m,k_m})] \\ \Gamma(e_1) = \tau_1 \ \ldots \ \Gamma(e_n) = \tau_n \\ \forall i \in \{1, \ldots, m\},\ \Gamma(o_{i,1}) = \tau_{i,1} \ \ldots \ \Gamma(o_{i,k_i}) = \tau_{i,k_i} \\ \{o_{i,1}, \ldots o_{i,k_i}\} \in \mathcal{O} \qquad \forall i, \forall j, \forall k_i,\ j \neq k_i\ o_{i,j} \neq o_{i,k_i} \qquad \lambda \in \{\lambda_1, \ldots, \lambda_m\}\end{array}}{\Sigma, \Gamma, \mathcal{O} \vdash p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m] \to \lambda}\ \text{WTPCall}$$

The inference rules for the αSmil statements representing calls to built-in predicates are detailed in Table 4.7.

TABLE 4.7 – Well-Typed Statements

$$\frac{\Gamma(e_1) = \Gamma(e_2) \qquad \lambda \in \{\mathsf{true}, \mathsf{false}\}}{\Sigma, \Gamma, \mathcal{O} \vdash e_1 = e_2 \to \lambda} \text{ WTEQUALS}$$

$$\frac{\Gamma(o) = \Gamma(e) \qquad o \in \mathcal{O}}{\Sigma, \Gamma, \mathcal{O} \vdash o := e \to \mathsf{true}} \text{ WTASGN}$$

$$\frac{}{\Sigma, \Gamma, \mathcal{O} \vdash \boldsymbol{nop} \to \mathsf{true}} \text{ WTNOP}$$

$$\frac{\begin{array}{c}\Gamma(r) = \boldsymbol{struct}\{f_1 : \tau_1, \ldots, f_n : \tau_n\} \\ \Gamma(e_1) = \tau_1 \ldots \Gamma(e_n) = \tau_n \qquad r \in \mathcal{O}\end{array}}{\Sigma, \Gamma, \mathcal{O} \vdash r := \{e_1, \ldots, e_n\} \to \mathsf{true}} \text{ WTRECNEW}$$

$$\frac{\begin{array}{c}\Gamma(r) = \boldsymbol{struct}\{f_1 : \tau_1, \ldots, f_n : \tau_n\} \\ \Gamma(o_1) = \tau_1 \ldots \Gamma(o_n) = \tau_n \qquad \forall i, o_i \in \mathcal{O} \qquad \forall i \neq j, o_i \neq o_j\end{array}}{\Sigma, \Gamma, \mathcal{O} \vdash \{o_1, \ldots, o_n\} := r \to \mathsf{true}} \text{ WTRECALL}$$

$$\frac{\Gamma(r) = \boldsymbol{struct}\{f_1 : \tau_1, \ldots, f_i : \tau_i, \ldots, f_n : \tau_n\} \qquad \Gamma(o) = \tau_i \qquad o \in \mathcal{O}}{\Sigma, \Gamma, \mathcal{O} \vdash o := r.f_i \to \mathsf{true}} \text{ WTRECGET}$$

$$\frac{\begin{array}{c}\Gamma(r) = \Gamma(r') = \boldsymbol{struct}\{f_1 : \tau_1, \ldots, f_i : \tau_i, \ldots, f_n : \tau_n\} \\ \Gamma(e) = \tau_i \qquad r' \in \mathcal{O}\end{array}}{\Sigma, \Gamma, \mathcal{O} \vdash r' := \{r \ \boldsymbol{with}\ f_i = e\} \to \mathsf{true}} \text{ WTRECSET}$$

$$\frac{\begin{array}{c}\Gamma(r') = \Gamma(r'') = \boldsymbol{struct}\{g_1 : \tau_1, \ldots, g_n : \tau_n\} \\ \lambda \in \{\mathsf{true}, \mathsf{false}\} \qquad \{f_1, \ldots, f_k\} \subseteq \{g_1, \ldots, g_n\}\end{array}}{\Sigma, \Gamma, \mathcal{O} \vdash r' = \langle f_1, \ldots, f_k \rangle r'' \to \lambda} \text{ WTRECEQ}$$

$$\frac{\begin{array}{c}\Gamma(v) = \boldsymbol{variant}[C_1 : \tau_1 | \ \ldots | \ C_p : \tau_p | \ \ldots | \ C_n : \tau_n] \\ \Gamma(e) = \tau_p \qquad v \in \mathcal{O}\end{array}}{\Sigma, \Gamma, \mathcal{O} \vdash v := C_p[e] \to \mathsf{true}} \text{ WTVARCONS}$$

$$\frac{\begin{array}{c}\Gamma(v) = \boldsymbol{variant}[C_1 : \tau_1 | \ \ldots | \ C_p : \tau_p | \ \ldots | \ C_n : \tau_n] \\ \Gamma(o_p) = \tau_p \qquad o_p \in \mathcal{O}\end{array}}{\Sigma, \Gamma, \mathcal{O} \vdash \boldsymbol{switch}(v) \ \boldsymbol{as}\ [o_1 | \ldots | o_n] \to \lambda_{C_p}} \text{ WTVARSWITCH}$$

$$\frac{\Gamma(v) = \textbf{\textit{variant}}[D_1 : \tau_1 |\ \ldots |\ D_m : \tau_m] \quad \{C_1, \ldots, C_k\} \subseteq \{D_1, \ldots, D_m\} \qquad \lambda \in \{\mathsf{true}, \mathsf{false}\}}{\Sigma, \Gamma, \mathcal{O} \vdash v \in \{C_1, \ldots, C_k\} \to \lambda} \text{ WTVarPos}$$

$$\frac{\Gamma(a) = \textbf{\textit{arr}}^{\tau_i}\langle\tau\rangle \qquad \lambda \in \{\mathsf{true}, \mathsf{false}\} \qquad \Gamma(i) = \tau_i \qquad \Gamma(o) = \tau \qquad o \in \mathcal{O}}{\Sigma, \Gamma, \mathcal{O} \vdash o := a[i] \to \lambda} \text{ WTAGet}$$

$$\frac{\Gamma(a') = \Gamma(a) = \textbf{\textit{arr}}^{\tau_i}\langle\tau\rangle \quad \lambda \in \{\mathsf{true}, \mathsf{false}\} \qquad \Gamma(i) = \tau_i \qquad \Gamma(e) = \tau \qquad a' \in \mathcal{O}}{\Sigma, \Gamma, \mathcal{O} \vdash a' := [a \textbf{ \textit{with}} \ i = e] \to \lambda} \text{ WTASet}$$

The well-typedness of statements plays an important role with respect to the statements' interpretation, as we will show in the next section. It is also essential for the well-typedness and well-formedness of dependency and correlation summaries that will be presented in the following chapters.

The control flow graph $G_p = (N, \mathcal{E})$ of a predicate p is well-typed if any edge labeled with $(s, \lambda) \in \mathcal{E}$ is well-typed:

$$\frac{\forall (s, \lambda) \in \mathcal{E},\ \Sigma, \Gamma, \mathcal{O} \vdash s \to \lambda}{\Sigma, \Gamma, \mathcal{O} \vdash G_p = (N, \mathcal{E})} \text{ WTCfg}$$

Figure 4.3 – Well-Typed Control Flow Graph

4.4 Operational Semantics of αSmil Statements

This section presents the *structural operational semantics* (Nielson, Nielson, and Hankin, 1999; Plotkin, 2004) of the αSmil language. Sometimes also called the *small step operational semantics*, this allows reasoning about intermediate stages in a program's execution and emphasizes the individual steps of the computation.

Types We take T_0 to be the universe of primitive types $\tau_0 \in T_0$. Structures, variants and associative arrays are defined inductively. Structures are *finite labeled products* of types. They are a generalization of the Cartesian product. Variants are *finite labeled disjoint unions* of several types τ. Two types are equal when they are pointwise equal.

Semantic Values. For each type τ we define the set $\mathbb{D}_\tau$ of semantic values of that type. For each primitive type $\tau_0 \in T_0$, we suppose a given $\mathbb{D}_{\tau_0}$. Other semantic values are defined inductively as shown below.

Definition 4.4.1. Semantic Values $\mathbb{D}_\tau$.

$$\mathbb{D}_{\textbf{\textit{struct}}\{f_1:\tau_1,\ldots,f_n:\tau_n\}} = \{\{f_1 = v_1, \ldots, f_n = v_n\} |\ \forall i, v_i \in \mathbb{D}_{\tau_i}\}$$

$$\mathbb{D}_{\textbf{\textit{variant}}[C_1:\tau_1|\ \ldots|\ C_n:\tau_n]} = \biguplus_{1\leq i\leq n}\{C_i[v] |\ v \in \mathbb{D}_{\tau_i}\}, \text{where} \biguplus \text{ is the disjoint union}$$

$$\mathbb{D}_{\textbf{\textit{arr}}^{\tau_i}\langle\tau\rangle} = \{(\mathcal{P}, (v_k)_{k\in\mathcal{P}}) |\ \mathcal{P} \subseteq \mathbb{D}_{\tau_i}, \forall k \in \mathcal{P}, v_k \in \mathbb{D}_\tau\}.$$

In αSmil, arrays are partial. In a semantic value belonging to $\mathbb{D}_{\textbf{\textit{arr}}^{\tau_i}\langle\tau\rangle}$, $\mathcal{P}$ denotes the domain of valid indices for the array.

Two values of the same type are equal when they are pointwise equal.

Traditionally, in operational semantics one is interested in how the state is modified during the execution of a statement. αSmil has no concept of state *per se*; what is essential is the evaluation of variables in different environments or *semantic contexts*. To emphasize this idea, we define a *valuation* or *environment* $E \in \mathscr{E}$, as a mapping from variables to semantic values.

Definition 4.4.2. Valuation or environment E.

$$E : \mathcal{V} \to \mathbb{D}.$$

Two valuations E and E' are equal if they are mapping the same set of variables to semantic values that are *pointwise* equal:

$$E = E' \iff \forall v \in \mathcal{V}, E(v) = E'(v).$$

Given a typing environment Γ, a valuation E is well-typed if the value mapped to any variable $v \in Dom(E)$ is of the appropriate type $\Gamma(v)$. We denote this by $\Gamma \vdash E$ and show it in [WTEnv]:

$$\frac{\forall v \in E,\ E(v) \in \mathbb{D}_{\Gamma(v)}}{\Gamma \vdash E}\ \text{WTEnv}$$

Definition 4.4.3. A *configuration* $\langle E, [s]\rangle$ of the semantics is a pair consisting of a valuation and a statement.

Definition 4.4.4. The *transitions* of the semantics are of the form:

$$\langle E, [s]\rangle \xrightarrow{\lambda} E'.$$

They express how the configuration is changed by *one step* of computation, occuring when executing a statement s that exits with label λ. The exit label yielded by the statement's execution uniquely determines the statement that will be executed next. The change of the valuation is recorded in the resulting valuation E'. We write

$E\,[x \to v]$ for the valuation that is identical to E except that x is mapped to the value v. We say that E is *extended* with $x \to v$ and formally we define it as shown below.

Definition 4.4.5. Extend E with $x \to v$.

$$(E\,[x \to v])(y) = \begin{cases} v & \text{if } x = y \\ E(y) & \text{otherwise} \end{cases}$$

Extending a valuation E with multiple mappings $\bar{x} \to \bar{v}$, consists in applying the extension in a left-associative fashion. In the following we will omit parentheses for such extensions, thus denoting:

$$(\ldots((E\,[x_1 \to v_1])[x_2 \to v_2])\ldots)[x_n \to v_n]$$

as:

$$E\,[x_1 \to v_1]\,[x_2 \to v_2]\ldots[x_n \to v_n].$$

An interpretation $I \in \mathscr{I}$ for a predicate is defined as a mapping from a predicate and an initial environment to an output environment and an exit label.

Definition 4.4.6. Predicate Interpretation $I \in \mathscr{I}$.

$$I : \mathscr{P} \times \mathscr{E} \to \mathscr{E} \times \mathscr{L}.$$

The initial environment is a mapping between the predicate's formal arguments and their effective values. The output environment is a mapping between the predicate's formal output arguments and their effective values after executing the predicate.

The detailed definition of the semantics of generic statements is described below in Table 4.8. The first clause, [*nop*] constitutes an axiom, as it has no premises. It states that the ***nop*** statement executes in one step, yielding the exit label true without *extending* the valuation E. The semantics of equality tests is given by two inference rules, [*equalT*] and [*equalF*], one for each of the statement's possible exit labels. A call to the built-in predicate = will exit with label true if and only if the valuations of its arguments e_1 and e_2 are equal (clause [*equalT*]). Otherwise, the statement will exit with label false (clause [*equalF*]). In both cases, the statement leaves the valuation E unchanged. The semantics of an assignment is given by the [*asgn*] clause: the statement always yields the exit label true and extends the valuation E with o mapped to the value $E(e)$ of e.

Table 4.8 – The Structural Operational Semantics of αSmil Generic Statements

[*nop*]	$\dfrac{}{\langle E, [\boldsymbol{nop}]\rangle \xrightarrow{\text{true}} E}$
[*equalT*]	$\dfrac{E(e_1) = E(e_2)}{\langle E, [e_1 = e_2]\rangle \xrightarrow{\text{true}} E}$

$$[equalF] \quad \frac{E(e_1) \neq E(e_2)}{\langle E, [e_1 = e_2] \rangle \xrightarrow{\mathsf{false}} E}$$

$$[asgn] \quad \frac{E' = E\,[o \rightarrow E(e)]}{\langle E, [o := e] \rangle \xrightarrow{\mathsf{true}} E'}$$

The semantics of structure-related statements is given in the Table 4.9. The creation of a structure always yields the exit label true, as indicated by the [*recNew*] clause and it extends the valuation E by mapping the resulting output variable r to the structural value obtained by mapping every field f_i to the value $E(e_i)$ of the corresponding e_i arguments. The destructuring of a structure r extends the valuation E by mapping every output o_i to the corresponding value $E(v_i)$ of the f_i field of r. The statement always exits with true. The valuation E' obtained after executing an access to a given field f_i of a structure r is an extension of E where the output o is mapped to the corresponding value of r's f_i field in E. The semantics of a field update is given by the clause [*recSet*]. This statement extends the valuation E by mapping the output structure r' to a new value, where the updated field f_i is mapped to the value of e in E and every other field is mapped to the same value it had in E. Finally, the last two clauses correspond to a partial structure equality test. As shown by [*recEqualsT*], the statement yields the exit label true if and only if the values of every field g_i in the given set of fields are equal for r and r' in E. Otherwise, the statement yields the label false. In both cases, the valuation E remains unchanged.

TABLE 4.9 – Operational Semantics of αSmil Structure-Related Statements

$$[recNew] \quad \frac{E' = E\,[r \rightarrow \{f_1 = E(e_1), \ldots, f_i = E(e_i), \ldots, f_n = E(e_n)\}]}{\langle E, [r := \{e_1, \ldots, e_n\}] \rangle \xrightarrow{\mathsf{true}} E'}$$

$$[recAll] \quad \frac{\begin{array}{c} E(r) = \{f_1 = v_1, \ldots, f_n = v_n\} \\ E' = E\,[o_1 \rightarrow v_1]\,[o_2 \rightarrow v_2] \ldots [o_n \rightarrow v_n] \qquad \forall i, j,\ i \neq j,\ o_i \neq o_j \end{array}}{\langle E, [\{o_1, \ldots, o_n\} := r] \rangle \xrightarrow{\mathsf{true}} E'}$$

$$[recGet] \quad \frac{\begin{array}{c} E(r) = \{f_1 = v_1, \ldots, f_i = v_i, \ldots, f_n = v_n\} \\ E' = E\,[o \rightarrow v_i] \end{array}}{\langle E, [o := r.f_i] \rangle \xrightarrow{\mathsf{true}} E'}$$

$$[recSet] \quad \frac{\begin{array}{c} E(r) = \{f_1 = v_1, \ldots, f_i = v_i, \ldots, f_n = v_n\} \\ E' = E\,[r' \rightarrow \{f_1 = v_1, \ldots, f_i = E(e), \ldots, f_n = v_n\}] \end{array}}{\langle E, [r' := \{r\ \textbf{\textit{with}}\ f_i = e\}] \rangle \xrightarrow{\mathsf{true}} E'}$$

$$[recEqualsT] \quad \frac{E(r') = \{f_1 = v_{f_1}, \ldots, f_n = v_{f_n}\} \quad E(r'') = \{f_1 = w_{f_1}, \ldots, f_n = w_{f_n}\} \quad \{g_1, \ldots, g_k\} \subseteq \{f_1, \ldots, f_n\} \quad v_{g_i} = w_{g_i}, \forall i \in \{1, \ldots, k\}}{\langle E, [r' = \langle g_1, \ldots, g_k \rangle r'']\rangle \xrightarrow{\mathsf{true}} E}$$

$$[recEqualsF] \quad \frac{E(r') = \{f_1 = v_{f_1}, \ldots, f_n = v_{f_n}\} \quad E(r'') = \{f_1 = w_{f_1}, \ldots, f_n = w_{f_n}\} \quad \{g_1, \ldots, g_k\} \subseteq \{f_1, \ldots, f_n\} \quad \exists i, i \in \{1, \ldots, k\},\ v_{g_i} \neq w_{g_i}}{\langle E, [r' = \langle g_1, \ldots, g_k \rangle r'']\rangle \xrightarrow{\mathsf{false}} E}$$

Table 4.10 details the semantics of variant-related statements. As indicated by the [*varCons*] clause, the construction of a variant v with a constructor C_p always yields the exit label true. The obtained valuation E' is an extension of E, where the value of v is obtained by applying the constructor C_p to the argument's value, $E(e)$. A variant switch exits with the label λ_{C_i}, if the value of v in E has been constructed with the C_i constructor. The valuation E' obtained after executing the statement is an extension of E whereby the corresponding output o_i is mapped to the value of the C_i constructor's argument, $E(e)$. The last two clauses, [*varPossibleT*] and [*varPossibleF*], indicate the semantics of a variant possible check and correspond to the statement's possible exit labels. The statement will yield the label true only if the value of v in E has been obtained with a constructor D that is a member of the given set of constructors $\{C_1, \ldots, C_k\}$. Otherwise, the false label will be returned. In both cases the valuation remains unchanged.

TABLE 4.10 – Operational Semantics of αSmil Variant-Related Statements

$$[varCons] \quad \frac{E' = E\,[v \rightarrow C_p[E(e)]]}{\langle E, [v := C_p[e]]\rangle \xrightarrow{\mathsf{true}} E'}$$

$$[varSwitch] \quad \frac{E(v) = C_i[e] \quad E' = E\,[o_i \rightarrow E(e)]}{\langle E, [\textbf{\textit{switch}}(v)\ \textbf{\textit{as}}\ [o_1|\ldots|o_n]]\rangle \xrightarrow{\lambda_{C_i}} E'}$$

$$[varPossibleT] \quad \frac{E(v) = D[e] \quad D \in \{C_1, \ldots, C_k\}}{\langle E, [v \in \{C_1, \ldots, C_k\}]\rangle \xrightarrow{\mathsf{true}} E}$$

$$[varPossibleF] \quad \frac{E(v) = D[e] \quad D \notin \{C_1, \ldots, C_k\}}{\langle E, [v \in \{C_1, \ldots, C_k\}]\rangle \xrightarrow{\mathsf{false}} E}$$

Table 4.11 describes the semantics of array-related statements. Each array-related statement has two corresponding clauses, one for each of the Boolean exit labels. Accessing an array's element yields the exit label true if the given index i is a valid index. The resulting valuation E' is extended by mapping the output o to the value in E of the array's i-th element. Otherwise, when the given index i is invalid, as indicated by the [*arrGetF*] clause, the statement yields the label false and leaves the valuation unmodified. The semantics of an array update is given by the [*arrSetT*] and [*arrSetF*] clauses. If the given index i is valid, the exit label true is yielded and the resulting valuation is obtained by extending E with a' whose i-th element's value is the value of e in the initial valuation E. The values of all other elements of a' are the ones found in E for the elements of a. On the contrary, if the given index i is invalid, the valuation remains unchanged and the label false is yielded.

TABLE 4.11 – Operational Semantics of αSmil Array-Related Statements

$$[arrGetT] \quad \frac{E(a) = (\mathcal{P}, (v)_k), \qquad E(i) \in \mathcal{P} \qquad E' = E\left[o \to v_{E(i)}\right]}{\langle E, [o := a[i]] \rangle \xrightarrow{\mathsf{true}} E'}$$

$$[arrGetF] \quad \frac{E(a) = (\mathcal{P}, (v)_k), \qquad E(i) \notin \mathcal{P}}{\langle E, [o := a[i]] \rangle \xrightarrow{\mathsf{false}} E}$$

$$[arrSetT] \quad \frac{E(a) = (\mathcal{P}, (v)_k) \qquad E(i) \in \mathcal{P} \qquad E\left[a' \to (\mathcal{P}, (w)_k), w_k = \begin{cases} E(e) & \text{if } k = E(i) \\ v_k & \text{otherwise} \end{cases}\right]}{\langle E, [a' := [a \ \textbf{\textit{with}} \ i = e]] \rangle \xrightarrow{\mathsf{true}} E'}$$

$$[arrSetF] \quad \frac{E(a) = (\mathcal{P}, (v)_k) \qquad E(i) \notin \mathcal{P}}{\langle E, [a' := [a \ \textbf{\textit{with}} \ i = e]] \rangle \xrightarrow{\mathsf{false}} E}$$

The semantics of a generic predicate call $p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]$ is captured by the [*pCall*] inference rule shown in Table 4.12. Interpreting the predicate p in the context of its arguments' values in the valuation E, yields a label λ_i and a mapping between its formal output arguments and their resulting values $v_{i,j}$. The resulting evaluation E' is obtained by extending E with the output variables $o_{i,j}$ mapped to the corresponding $v_{i,j}$.

The interpretation of a statement is well-typed with respect to a signature if and only if every tuple in the interpretation is well-typed, i.e. if it has the expected number of inputs, with the adequate types, and an adequate label with well-typed outputs as

well. Furthermore, it has to be *total*, i.e. for every well-typed tuple of inputs, there exists a label and some outputs that match in the interpretation.

TABLE 4.12 – Semantics of a Predicate Call

$$\frac{\begin{array}{c}\Sigma(p) = p(x_1 : \tau_1, \ldots, x_n : \tau_n)[\lambda_1 : (\bar{\tau}_1 : \bar{y}_1)| \ \ldots \\ \ldots | \ \lambda_i : (\tau_{i,1} : y_{i,1}, \ldots, \tau_{i,k_i} : y_{i,k_i})| \ \ldots | \ \lambda_m : (\bar{\tau}_m : \bar{y}_m)] \\ I(p, inputs) = (outputs, \lambda_i) \qquad inputs(x_l) = E(e_l), \forall l \in \{1, \ldots, n\} \\ outputs(y_{i,1}) = v_{i,1}, \ldots, outputs(y_{i,k_i}) = v_{i,k_i} \\ E' = E\,[o_{i,1} \to v_{i,1}] \ldots [o_{i,k_i} \to v_{i,k_i}]\end{array}}{\langle E, [p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]]\rangle \xrightarrow{\lambda_i} E'} \text{ pCALL}$$

Definition 4.4.7. *Subject Reduction Property.*

The interpretation of a well-typed statement given well-typed interpretations for the external predicate calls, preserves the fact that the valuation is well-typed:

$$\forall\ \Gamma, E, s, \lambda, \Sigma,\ (\Gamma \vdash E) \wedge (\Sigma, \Gamma, \mathcal{O} \vdash s \to \lambda) \wedge (\langle E, [s]\rangle \xrightarrow{\lambda} E') \implies \Gamma \vdash E'.$$

Definition 4.4.8. *The Progress Property.*

A well-typed statement in a well-typed environment can always be interpreted to some label and outputs:

$$\forall\ E, \Gamma, \Sigma, s,\ (\Gamma \vdash E) \wedge (\Sigma, \Gamma, \mathcal{O} \vdash s \to \lambda) \implies \exists \lambda', E',\ \langle E, [s]\rangle \xrightarrow{\lambda'} E'.$$

The well-typedness of an interpretation, as well as the subject reduction and progress properties have been formally proven in Coq by Stéphane Lescuyer.

Chapter 5

Dependency Analysis for Functional Specifications

> ... like islands in the sea, separate on the surface but connected in the deep.
>
> William James

Algebraic data types (structures and variants) and associative arrays are fundamental building blocks for representing, grouping and handling complex data efficiently. However, as argued in Chapter 1, operations manipulating them are rarely concerned with the entire compound input data structure. Frequently, they depend only on a limited subset of their input. Complete specifications or *contracts* (Meyer, 1997) of such operations will not only stipulate that the output possesses a certain property (Borgida, Mylopoulos, and Reiter, 1993; Polikarpova et al., 2013), but will also include their *frame conditions* (Borgida, Mylopoulos, and Reiter, 1995), i.e. the parts of the input on which they operate. Such conditions facilitate reasoning locally without overlooking the global picture: if a property P is known to hold at a certain point in the program where a predicate p is called, P still holds after the call to p, provided that the (sub)structures on which P depends are disjoint from the (sub)structures that might be modified according to p's frame condition (Banerjee and Naumann, 2014). Though intuitively easy, specifying and proving the preservation of logical properties for the unmodified part is a particular manifestation of the *frame problem* (McCarthy and Hayes, 1969; Leavens, Leino, and Müller, 2007) – a notoriously cumbersome task in formal software verification, imposing unnecessary manual effort (Meyer, 2015).

One of the challenges of addressing this problem and thereby simplifying the verification of certain preserved properties is to determine the input fragments on which these properties depend, i.e. their *footprint* (Distefano, O'Hearn, and Yang, 2006) or, to a *first approximation*, their *read effects* (Feijs and Jonkers, 1992; Greenhouse and Boyland, 1999; Clarke and Drossopoulou, 2002). While specifications sometimes include the *write effects* (Clarke and Drossopoulou, 2002) of an operation through *modifies* clauses (Guttag et al., 1993b), read effects are usually not specified explicitly even though this information can be useful for reasoning about an operation's results. The purpose of the dependency analysis presented in this chapter is to take a step forward in

this direction and to detect such information automatically. More precisely, our analysis is a static dependency analysis for the αSmil language (presented in Chapter 4) that computes a conservative approximation of the input fragments on which the operations depend.

Dependence and liveness analyses are traditionally used in the compilation realm, for code optimization (Kennedy, 1978), dead code elimination (Knoop, Rüthing, and Steffen, 1994; Wand and Siveroni, 1999; Liu and Stoller, 2003), program slicing (Weiser, 1984; Tip, 1995; Reps and Turnidge, 1996; Castillo et al., 2008) or compile-time garbage collection (Jones and Métayer, 1989; Park and Goldberg, 1992; Wand and Clinger, 1998). In contrast to the vast majority of static analyses that are meant to be used strictly on code and in an essentially purely automatic setting, our analysis is thought of as a companion tool to be exploited in the middle of *interactive* program verification and it is designed to be used on programs as well as on specifications.

5.1 Dependency Analysis in a Nutshell

In a nutshell, our dependency analysis targets the delimitation of the input subset on which the output depends, in the context of an operation with a compound input. We define *dependency* as the observed part of a structured domain and strive to obtain type-sensitive results, distinguishing between the subelements of arrays and algebraic data types and capturing the dependency specific to each. The targeted results are meant to mirror – in terms of dependency – the layered structure of compound data types. Furthermore, the *dependency analysis* must work with *conservative approximations* and it must guarantee that what is marked as not needed is definitely not needed, i.e. it is irrelevant for the obtained output.

In the classification of Hind (Hind, 2001), our dependency analysis is a *flow-sensitive*, *field-sensitive*, *interprocedural* analysis that handles associative arrays, structures and variant data types. Specific dependency results are computed for each of the possible execution scenarios, i.e. for each exit label. Thus, our analysis also shows a form of *path-sensitivity* (Hind, 2001). However, we favour the term *label-sensitivity* to describe this characteristic, as it seems more appropriate applied to our case and the language we are working with.

Our dependency analysis targets complex transition systems in general, and operating systems and microkernels in particular. These are characterized by states defined by complex compound data structures and by transitions, i.e. state changes, that map an input state to an output state. Automatically proving the preservation of invariants concerning only subelements of the state, i.e. fields, array cells, etc., that have not been altered by a transition in the system would considerably diminish the number of proof obligations. The first step towards achieving this goal consists in automatically detecting dependency summaries and the minimum relevant input information for producing certain outputs.

As mentioned, our analysis targets fine-grained dependency summaries for arrays, structures and variants, expressed at the level of their subelements. For variants,

besides capturing the specific dependency on each constructor and its arguments, we argue that additional, relevant information can be computed, regarding the subset of possible constructors at a given program point. This is not dependency information *per se* but it enriches the footprint of a predicate with useful information. Together with the dependency information, this additional information about constructors is meant to answer the same question, namely, what fragments of the input influence the output, from a different, albeit related point of view. Therefore, we are simultaneously performing a *possible-constructors* analysis. This has an impact on the defined abstract dependency type, making it more complex, as we will see in the following section. The *possible-constructors* analysis could be performed separately, as a stand-alone analysis. By performing the two analyses simultaneously, we lose some of the precision that would be attained if the two were performed separately, but we reduce overhead and present relevant information in a unified manner.

Designing the analysis as a tool to be used in the context of interactive program verification, on both code and specifications, has led to specific traits. One of them concerns the treatment of arrays. In contrast to dependence and liveness analyses used for code optimizations (Gross and Steenkiste, 1990), which require precision for every array cell, we compute dependency information referring to all cells of the array or to all but one cell, for which an exceptional dependency is computed. In practice, a considerable number of relevant properties and operations involving arrays fall into this spectrum.

In the following subsection, in order to better illustrate the problem that our analysis addresses, we briefly present two examples of αSmil predicates manipulating structures, variants and arrays and describe the dependency information that we are targeting.

5.1.1 Targeted Dependency Information

To present the envisioned dependency results, we consider two αSmil predicates, `thread` and `start_address`, whose control flow graphs and implementations are shown below. Both predicates manipulate inputs of type `process`, introduced in Section 3.1.5 (on page 49) and shown in Figure 5.2. Internally, they handle values of type `thread` and `memory_region`, respectively, described in Section 3.1.5 (on page 48) as well and shown below in Figure 5.1.

```
type memory_region = {
 // Start address
 start : int;
 // Region length
 length : int
}
```

```
type thread = {
 // Identifier
 id : int;
 // Current state
 crt_state : state;
 // Stack
 stack : memory_region
}
```

FIGURE 5.1 – Example Data Types – Thread and Memory Region

```
type option<A> =
  | None
  | Some (A a)
```

```
type process = {
 // Array of associated threads
 threads : array<option<thread>>;
 // Internal id
 pid : int;
 // Currently running thread
 crt_thread : int;
 // Address space
 adr_space : address_space
}
```

FIGURE 5.2 – Input Type – Process

The first predicate, `thread`, having the control flow graph shown in Figure 5.4 and whose implementation is shown in Figure 5.3, receives a process `p` and an index `i` as inputs. It reads the `i`-th element in the `threads` array of the input process `p`. If this element is active, then the predicate exits with the label **true** and outputs the corresponding thread `ti`. Otherwise, it exits with the label `None` and no output is generated.

```
predicate thread(process p, int i)
 -> [true: thread ti|None|oob]
{{array<option<thread>> th, option<thread> tio}} {
  th := p.threads : [true -> 1];
  tio := th[i] : [true -> 2, false -> 5];
  switch (tio) as [ |ti] : [None -> 4, Some -> 3];
  [true];
  [None];
  [oob]
}
```

FIGURE 5.3 – Predicate `thread` – Implementation

Our dependency analysis should be able to distinguish between the different exit labels of the predicate. For the label **true** for instance, it should detect that only the field `threads` is read by the predicate, while all others are irrelevant to the result. Furthermore, it should detect that for the `threads` array of the input `p` only the `i`-th element is inspected. Additionally, since we are considering the label **true**, the `i`-th element is necessarily an *active* thread, indicated by the constructor `Some`. The other constructor, `None`, is *impossible* for this execution scenario. On the contrary, for the exit label `None`, the constructor `Some` is impossible. For the exit label `oob`, nothing but the index `i` and the "support" or "length" of the associated `threads` array is read. The targeted dependency results for the predicate `thread` are depicted in Figure 5.5.

The second predicate, `start_address`, whose control flow graph is shown in Figure 5.6, receives a process `p` and an index `j` as inputs and finds the start address of

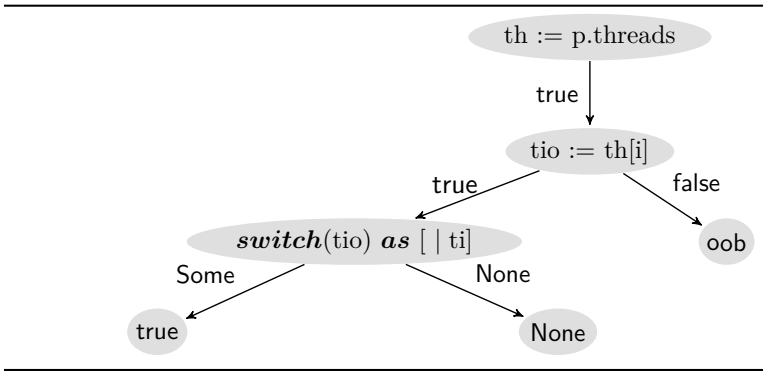


FIGURE 5.4 – G_{thread} – Control Flow Graph of Predicate `thread`

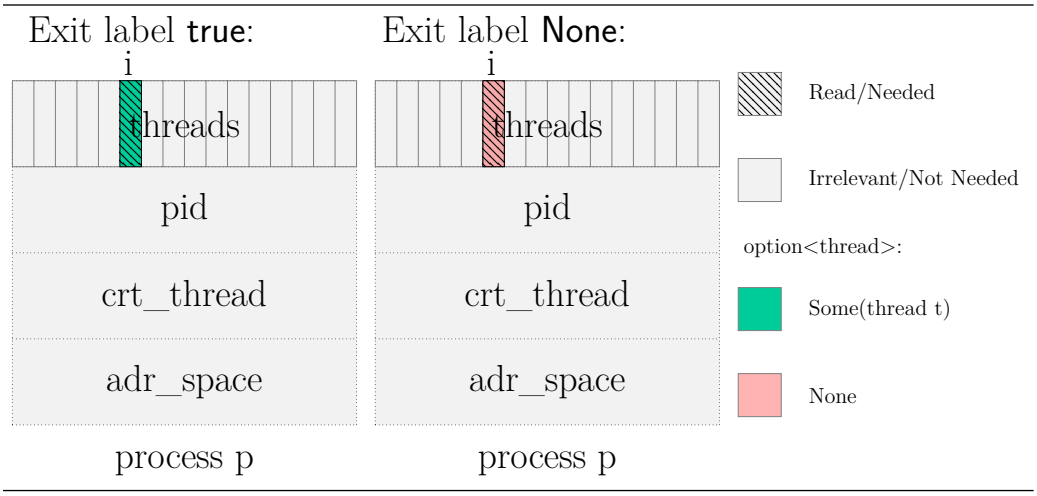


FIGURE 5.5 – Targeted Dependency Results for Predicate `thread`

the stack corresponding to an active thread. It makes a call to the predicate `thread`, thus reading the `j`-th element of the `threads` array of its input process. If this is an active element, it further accesses the field `stack`, from which it only reads the start address `start`. Otherwise, if the element is inactive, the predicate forwards the exit label `None` of the called predicate `thread` and generates no output. When given an invalid index `i`, the predicate exits with label `oob`. The predicate's implementation is shown in Figure 5.7.

The dependency information for this predicate should capture the fact that on the **true** execution scenario, only the field `start` of the input's `j`-th associated thread is read. Furthermore, the only possible constructor on this execution path is the `Some` constructor. On the contrary, for the `None` execution scenario the only possible constructor is the `None` constructor. The targeted dependency results for the `start_address` predicate are depicted in Figure 5.8. We remark that for the `oob` execution scenario, only the "support" or "length" of the `threads` array is read.

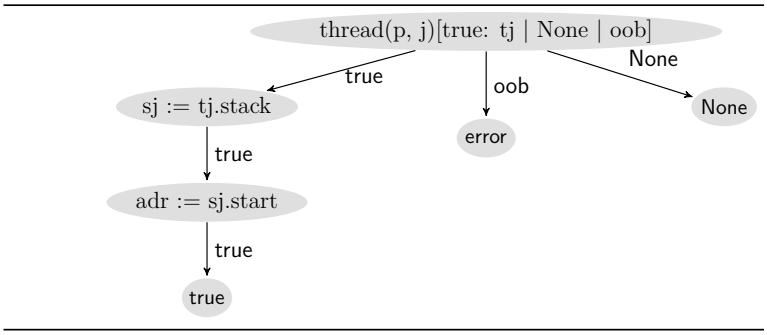


FIGURE 5.6 – $G_{start\_address}$ – Control Flow Graph of Predicate `start_address`

```
predicate start_address(process p, int j)
 -> [true: int adr|None]
{{thread tj, memory_region sj}} {
  thread(p, j)[true: tj | None | oob] : [true -> 1,
   None -> 4, oob -> 5];
  sj := tj.stack : [true -> 2];
  adr := sj.start : [true -> 3];
  [true];
  [None];
  [error]
}
```

FIGURE 5.7 – Predicate `start_address` – Implementation

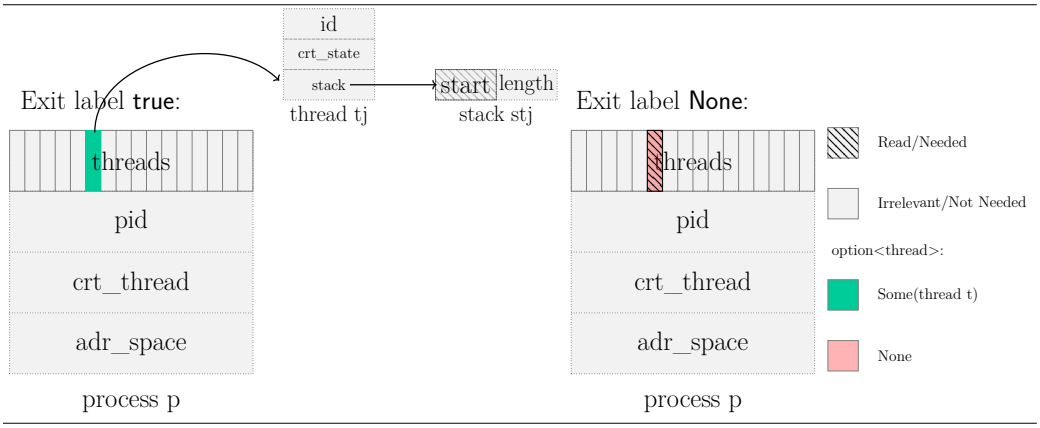


FIGURE 5.8 – Targeted Dependency Results for Predicate `start_address`

5.1.2 Outline

The rest of this chapter is focusing on technical details related to the dependency analysis. In Section 5.2 we present the abstract dependency domain. This is the fundamental building block on which our analysis relies in order to determine expressive dependency summaries. It is followed in Section 5.3 by an in-depth description of our analysis at an intraprocedural level, underlining the data-flow equations in Section 5.3.2 and explaining them by illustrating the step-by-step mechanism on an example in Section 5.3.3. A summary of the dependency analysis at an interprocedural level is given in Section 5.4. We illustrate the approach, underline its shortcomings on an example in Section 5.4.1, and discuss their origin in Section 5.4.2. Two different semantic interpretations of our dependency information are discussed in Section 5.5. In Section 5.6 we review and discuss approaches targeting information that is similar to our dependency summaries. Finally, in Section 5.7 we conclude and present some other potential applications of our dependency analysis, which are not confined to the field of interactive program verification.

5.2 Abstract Dependency Domain

The first step towards inferring expressive, type-sensitive results that capture the dependency specific to each subelement of an algebraic data type or an associative array, is the definition of an *abstract dependency domain* $\mathcal{D}$, that mimics the structure of such data types. The dependency domain $\delta \in \mathcal{D}$, shown below, is defined inductively from the three atomic cases — $\top$, $\oslash$ and $\bot$ — and mirrors the structure of the concrete types.

Definition 5.2.1. Dependency Domain $\delta \in \mathcal{D}$.

$\delta :=$	$\mid \top$	*Everything* – atomic case	(i)
	$\mid \oslash$	*Nothing* – atomic case	(ii)
	$\mid \bot$	*Impossible* – atomic case	(iii)
	$\mid \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}$	$f_1, \ldots, f_n$ fields	(iv)
	$\mid [C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m]$	$C_1, \ldots, C_m$ constructors	(v)
	$\mid \langle \delta \rangle$		(vi)
	$\mid \langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle$	i array index	(vii)

As reflected by the above definition, the dependency for atomic types is expressed in terms of the domain's atomic cases: $\top$ (least precise), denoting that *everything* is needed and $\oslash$, denoting that *nothing* is needed. The third atomic case $\bot$, denoting *impossible*, is introduced for the *possible constructors* analysis performed simultaneously, and is further explained below.

The dependency of a structure (iv) describes the dependency on each of its fields. For instance, revisiting our `thread` example from Section 5.1.1, we could express an over-approximation of the dependency information depicted for the process `p` in Figure 5.5

using the following dependency:

$$\{threads \mapsto \top;\ pid \mapsto \oslash;\ crt\_thread \mapsto \oslash;\ adr\_space \mapsto \oslash\}.$$

This captures the fact that all fields except the `threads` field are irrelevant, i.e. they are not read and *nothing* in their contents is needed. The dependency for the `threads` field is an over-approximation and expresses the fact that it is entirely necessary, i.e. *everything* in its value is needed for the result.

For arrays we distinguish between two cases, namely arrays with a general dependency applying to all of the cells given by (vi) and arrays with a general dependency applying to *all but one* exceptional cell, for which a specific dependency is known given by (vii). For instance, for the `threads` field of the previous example, the following dependency:

$$\langle \oslash \triangleright i :\ \top \rangle$$

would be a less coarse approximation, capturing the fact that only the `i`-th element of the associated `threads` array is needed, while all others are irrelevant.

For variants (v), the dependency is expressed in terms of the dependencies of their constructors, expressed in turn in terms of their arguments' dependencies. Thus, a constructor having a dependency mapped to $\oslash$ is one for which nothing but the *tag* has been read, i.e. its arguments, if any, are irrelevant for the execution. For instance, for the `i`-th element of the `threads` array of our previous example, the following dependency:

$$[Some \mapsto \top; None \mapsto \oslash]$$

would be a more precise approximation, when considering the exit label **true**. It is still an over-approximation as it expresses that both constructors are possible. The argument of the `Some` constructor is entirely read, while for `None` only the tag is read.

For variants, we want to take a step further and to also include the information that certain constructors cannot occur for certain execution paths. *Impossible*, the third atomic case — $\bot$ — is introduced for this purpose. As mentioned previously in Section 5.1, in order to obtain this additional information, we perform a "possible-constructors" analysis simultaneously, which computes for each execution scenario, the subset of possible constructors for a given value, at a given program point. All constructors that cannot occur on a given execution path are marked as being $\bot$. In contrast, constructors for which *only* the tag is read are marked as $\oslash$. The difference between $\bot$ and $\oslash$ can be illustrated by considering a polymorphic option type *option<A>*, having two constructors, $None$ and $Some(A\ val)$, respectively, and a Boolean predicate that pattern matches on an input of this type and returns *false* in the case of $None$ and *true* in the case of $Some$, unconditioned by the value *val* of its argument. For the *true* execution scenario, the dependency on the $Some$ constructor would be $\oslash$. The tag is read and it is decisive for the outcome, but the value of its argument *val* is completely irrelevant. The dependency on the $None$ constructor however would be $\bot$: the predicate can exit with label *true* if and only if the input matches against the $Some$ constructor. By distinguishing between these two cases we can not only distinguish the

input's subelements that have a direct impact on an operation's output, but additionally, we can also obtain a more detailed footprint that highlights the influence exerted by the input's "shape" on the operation's outcome.

For instance, for the `i`-th element of the `threads` array of our previous example, a dependency mapping the constructor `None` to $\bot$ would be a more precise approximation, when considering the label **true**. Taking into account all the discussed values, we can express the dependency depicted in Figure 5.5 for the label **true** as follows:

$$\left\{\begin{array}{lcl} \mathit{threads} & \mapsto & \langle \oslash \triangleright i : \ [\mathit{Some} \mapsto \top; \mathit{None} \mapsto \bot] \rangle \\ \mathit{pid} & \mapsto & \oslash \\ \mathit{crt\_thread} & \mapsto & \oslash \\ \mathit{adr\_space} & \mapsto & \oslash \end{array}\right\}.$$

We remark that $\top$, $\oslash$ and $\bot$ can apply to any type. For instance, $\top$ can be seen as a placeholder for data that is needed in its entirety. Structure, array or variant dependencies whose subelements are all entirely needed and thus, uniformly mapped to $\top$, are transformed to $\top$. The $\bot$ dependency is a placeholder for data that cannot occur on a certain execution scenario. A whole variant value is impossible if all its constructors are mapped to $\bot$. A whole structure or array is impossible if any of its subelements is impossible.

The $\bot$ atomic value is the lower bound of our domain and hence, the most precise value. The final abstract dependency is a closure of all these combined recursively. To give an intuition of the shape of our dependency lattice, we illustrate below in Figure 5.9 the Hasse diagram of the order relation between pairs of atomic dependency values. Intuitively, if the two analyses would be performed separately, the upper "diamond" shape would correspond to the dependency analysis, and the lower one to the possible-constructors analysis. The $\oslash$ element would be the lower bound for the dependency domain and the upper bound for the possible-constructors domain. By performing them simultaneously, $\bot$ becomes the domain's lower bound.

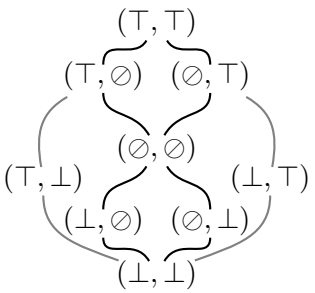


FIGURE 5.9 – Order Relation on Pairs of Atomic Dependencies

The partial order relation is denoted by $\sqsubseteq$ and defined as shown below.

Definition 5.2.2. Partial Order $\sqsubseteq$.

$$\sqsubseteq \ \subseteq \ \mathcal{D} \times \mathcal{D}.$$

TABLE 5.1 – $\sqsubseteq$ – Comparison of Two Domains

$$\frac{}{\delta \sqsubseteq \top}\ \textsc{Top} \qquad \frac{}{\bot \sqsubseteq \delta}\ \textsc{Bot}$$

$$\frac{\delta_1 \sqsubseteq \delta'_1 \quad \dots \quad \delta_n \sqsubseteq \delta'_n}{\{f_1 \mapsto \delta_1; \dots; f_n \mapsto \delta_n\} \sqsubseteq \{f_1 \mapsto \delta'_1; \dots; f_n \mapsto \delta'_n\}}\ \textsc{Str} \qquad \frac{\oslash \sqsubseteq \delta_1 \quad \dots \quad \oslash \sqsubseteq \delta_n}{\oslash \sqsubseteq \{f_1 \mapsto \delta_1; \dots; f_n \mapsto \delta_n\}}\ \oslash\textsc{Str}$$

$$\frac{\delta_1 \sqsubseteq \delta'_1 \quad \dots \quad \delta_n \sqsubseteq \delta'_n}{[C_1 \mapsto \delta_1; \dots; C_n \mapsto \delta_n] \sqsubseteq [C_1 \mapsto \delta'_1; \dots; C_n \mapsto \delta'_n]}\ \textsc{Var} \qquad \frac{\oslash \sqsubseteq \delta_1 \quad \dots \quad \oslash \sqsubseteq \delta_n}{\oslash \sqsubseteq [C_1 \mapsto \delta_1; \dots; C_n \mapsto \delta_n]}\ \oslash\textsc{Var}$$

$$\frac{\delta_{def} \sqsubseteq \delta'_{def}}{\langle \delta_{def} \rangle \sqsubseteq \langle \delta'_{def} \rangle}\ \textsc{ADef} \qquad \frac{\oslash \sqsubseteq \delta_{def}}{\oslash \sqsubseteq \langle \delta_{def} \rangle}\ \oslash\textsc{ADef}$$

$$\frac{\delta_{def} \sqsubseteq \delta'_{def} \qquad \delta_{exc} \sqsubseteq \delta'_{def}}{\langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle \sqsubseteq \langle \delta'_{def} \rangle}\ \text{AIA} \qquad \frac{\delta_{def} \sqsubseteq \delta'_{def} \qquad \delta_{def} \sqsubseteq \delta'_{exc}}{\langle \delta_{def} \rangle \sqsubseteq \langle \delta'_{def} \triangleright i : \ \delta'_{exc} \rangle}\ \text{AAI}$$

$$\frac{\delta_{def} \sqsubseteq \delta'_{def} \qquad \delta_{exc} \sqsubseteq \delta'_{exc}}{\langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle \sqsubseteq \langle \delta'_{def} \triangleright i : \ \delta'_{exc} \rangle}\ \text{AI} \qquad \frac{\oslash \sqsubseteq \delta_{def} \qquad \oslash \sqsubseteq \delta_{exc}}{\oslash \sqsubseteq \langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle}\ \oslash\text{AI}$$

$$\frac{\delta_{def} \sqsubseteq \delta'_{def} \qquad \delta_{exc} \sqsubseteq \delta'_{exc} \qquad \delta_{def} \sqsubseteq \delta'_{exc} \qquad \delta_{exc} \sqsubseteq \delta'_{def} \qquad i \neq j}{\langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle \sqsubseteq \langle \delta'_{def} \triangleright j : \ \delta'_{exc} \rangle}\ \text{AIJ}$$

It is used to compare dependencies and it is detailed in Table 5.1. We write $\delta_1 \sqsubseteq \delta_2$ and we read it as "a dependency δ_1 is more precise than another dependency δ_2" if it represents a smaller subset of a structural object, and if it allows at most as many constructors as δ_2. The greatest element is $\top$ (TOP) and $\bot$ is the least (BOT). Instances of identical structure and variant types are compared pointwise (STR, VAR). For arrays without known exceptional dependencies we compare the default dependencies applying to all array cells (ADEF). If exceptional dependencies are known for the same cell, these are additionally compared (AI). For arrays with known exceptional dependencies for different cells, we compare each dependency on the left-hand side with each one on the right-hand side (AIJ). The comparison of $\oslash$ with structures ($\oslash$STR), variants ($\oslash$VAR) and arrays ($\oslash$ADEF, $\oslash$AI) is a pointwise comparison between $\oslash$ and the dependency of each subelement.

5.2.1 Join and Reduction Operator

The *join* operation is denoted by $\vee$ and it is defined as shown below.

Definition 5.2.3. Join Operation $\vee$.

$$\vee \ : \ \mathcal{D} \times \mathcal{D} \ \rightarrow \ \mathcal{D}.$$

It is detailed in Table 5.2. Intuitively, the join of two dependencies is the union of the dependencies represented by the two. It is a *commutative* operation for which the undisplayed cases in Table 5.2 are defined by their symmetrical counterparts. The operation is *total*: joining incompatible domains such as a structure and a variant or two structures having different field identifiers, results in $\top$, the least precise value. Join is applied *pointwise* on each subelement; $\bot$ is its *identity* element and $\top$ is its *absorbing* element. Joining $\oslash$ and the dependency of a structure, variant or array is applied pointwise. The value obtained by joining δ and δ' is an *upper bound* of the two:

$$\delta \sqsubseteq \delta \vee \delta' \text{ and } \delta' \sqsubseteq \delta \vee \delta', \ \forall\, \delta, \delta' \in \mathcal{D}.$$

Defining the join of two dependencies corresponding to arrays is subtle. As shown in Table 5.1, we are allowing comparisons between dependencies corresponding to arrays with exceptions on different variables (rule AIJ); the join operation in this case amounts to joining the four different dependencies without keeping any of the two exceptions. We could have chosen to keep one of the known exceptional dependencies but this would have posed two problems: on one hand, the join operation would *not* be commutative, and, on the other hand, it is hard to predict how the exceptional dependencies would be used at the intraprocedural level and which of the two could potentially lead to a gain in precision. Thus, we adopted this design decision. A strategy possibly worth investigating in such cases would be to allow users to specify array cells of interest at specific program points. This user-supplied information could then be taken into consideration whenever joining array dependencies with two different known exceptional dependencies. Our current join approach for arrays can lead to *non-monotonic* approximations in join. This becomes visible when noting that for a

TABLE 5.2 – $\vee$ – Join Operation

δ'		δ''		$\delta' \vee \delta''$
$\top$	$\vee$	δ	$=$	$\top$
$\bot$	$\vee$	δ	$=$	δ
$\{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}$	$\vee$	$\{f_1 \mapsto \delta'_1; \ldots; f_n \mapsto \delta'_n\}$	$=$	$\{f_1 \mapsto \delta_1 \vee \delta'_1; \ldots; f_n \mapsto \delta_n \vee \delta'_n\}$
$\oslash$	$\vee$	$\{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}$	$=$	$\{f_1 \mapsto \oslash \vee \delta_1; \ldots; f_n \mapsto \oslash \vee \delta_n\}$
$[C_1 \mapsto \delta_1; \ldots; C_n \mapsto \delta_n]$	$\vee$	$[C_1 \mapsto \delta'_1; \ldots; C_n \mapsto \delta'_n]$	$=$	$[C_1 \mapsto \delta_1 \vee \delta'_1; \ldots; C_n \mapsto \delta_n \vee \delta'_n]$
$\oslash$	$\vee$	$[C_1 \mapsto \delta_1; \ldots; C_n \mapsto \delta_n]$	$=$	$[C_1 \mapsto \oslash \vee \delta_1; \ldots; C_n \mapsto \oslash \vee \delta_n]$
$\langle \delta_{def} \rangle$	$\vee$	$\langle \delta'_{def} \rangle$	$=$	$\langle \delta_{def} \vee \delta'_{def} \rangle$
$\oslash$	$\vee$	$\langle \delta_{def} \rangle$	$=$	$\langle \oslash \vee \delta_{def} \rangle$
$\langle \delta_{def} \rangle$	$\vee$	$\langle \delta'_{def} \triangleright i : \delta'_{exc} \rangle$	$=$	$\langle \delta_{def} \vee \delta'_{def} \triangleright i : \delta_{def} \vee \delta'_{exc} \rangle$
$\oslash$	$\vee$	$\langle \delta_{def} \triangleright i : \delta_{exc} \rangle$	$=$	$\langle \oslash \vee \delta_{def} \triangleright i : \oslash \vee \delta_{exc} \rangle$
$\langle \delta_{def} \triangleright i : \delta_{exc} \rangle$	$\vee$	$\langle \delta'_{def} \triangleright j : \delta'_{exc} \rangle$ $\begin{cases} i = j \\ i \neq j \end{cases}$	$=$	$\langle \delta_{def} \vee \delta'_{def} \triangleright i : \delta_{exc} \vee \delta'_{exc} \rangle$ $\langle \delta_{def} \vee \delta_{exc} \vee \delta'_{def} \vee \delta'_{exc} \rangle$
$\oslash$	$\vee$	$\oslash$	$=$	$\oslash$

monotonic join operation the following should hold:

$$\forall \delta, \delta', \rho,\ \delta \sqsubseteq \delta' \implies \delta \vee \rho \sqsubseteq \delta' \vee \rho \qquad \text{(i)}.$$

Considering:

$$\begin{array}{rcl} \rho & \equiv & \langle \rho_{def} \triangleright i :\ \rho_i \rangle \\ \delta & \equiv & \langle \delta_{def} \triangleright j :\ \delta_j \rangle \\ \delta' & \equiv & \langle \delta'_{def} \triangleright i :\ \delta'_i \rangle \text{ where } i \neq j \end{array}$$

the hypothesis $\delta \sqsubseteq \delta'$ is translated into the following constraints:

$$\delta_{def} \sqsubseteq \delta'_{def}, \quad \delta_{def} \sqsubseteq \delta'_i, \quad \delta_j \sqsubseteq \delta'_{def}, \quad \delta_j \sqsubseteq \delta'_i.$$

Applying (i) for these three dependencies, we obtain:

$$\langle (\delta_{def} \vee \delta_j) \vee (\rho_{def} \vee \rho_i) \rangle \sqsubseteq \langle \delta'_{def} \vee \rho_{def} \triangleright i :\ \delta'_i \vee \rho_i \rangle,$$

which holds if and only if both of the following inequalities hold:

$$\begin{array}{lcll} (\delta_{def} \vee \delta_j) & \vee & (\rho_{def} \vee \rho_i) & \sqsubseteq \delta'_{def} \vee \rho_{def} \\ (\delta_{def} \vee \delta_j) & \vee & (\rho_{def} \vee \rho_i) & \sqsubseteq \delta'_i \vee \rho_i \end{array}.$$

Considering, for instance,

$$\rho_i = \top,\ \rho_{def} \neq \top,\ \delta_{def} = \delta_j = \delta'_{def} = \bot,$$

a counterexample is found.

As a consequence of the *non-monotonic* approximations made for arrays (rule AIJ), the value obtained by joining two dependencies is an upper bound, not a *least upper bound*. We address this issue and indicate our solution in Section 5.3 (on page 94). We remark that we keep only one exceptional cell for array dependencies as in practice most operations manipulating arrays tend to either modify only one element or all of them. Logical properties on arrays generally have to hold for all elements. Keeping more than one exceptional dependency would be much more costly, and the additional cost would not necessarily be justified in practice. However, the join operation would be more straightforward and would not impose non-monotonic approximations.

Besides *join*, a *reduction operator* denoted by $\oplus$ has been defined as well.

Definition 5.2.4. Reduction Operator $\oplus$.

$$\oplus \ :\ \mathcal{D} \times \mathcal{D} \ \rightarrow\ \mathcal{D}.$$

This is a recursive, commutative, pointwise operation. Intuitively, this operator is introduced for taking advantage of the information additionally computed by the possible-constructors analysis that we perform simultaneously. Following the same execution path, the same constructors must be possible. The reduction operator is used in order to incorporate this additional information computed for constructors. The dependency

analysis can be seen as a *may* analysis, i.e. when combining the dependency information computed at two different points on the same execution path, the result must account for all dependencies computed at any of the two combined points. In contrast, the possible-constructors analysis can be seen as a *must* analysis, i.e. when combining information at two different points on the same execution path, it needs to keep facts that hold at both combined points. Thus, the reduction operator combines dependencies on the same execution path and consists in performing the intersection of constructors in the case of variants and the union of dependencies for all other types. The reduction operator's role will become more transparent after presenting the intraprocedural dependency analysis and the corresponding data-flow equations in Section 5.3. Its *identity* element is $\oslash$ and its *absorbing* element is $\bot$. The reduction operator between $\top$, and the dependency of a structure, variant or array is applied pointwise. Two instances of identical variant types are pointwise reduced. Similarly to join, the undisplayed cases in Table 5.3 are defined with respect to their symmetrical counterparts.

δ'		δ''		$\delta' \oplus \delta''$	
$\bot$	$\oplus$	δ	$=$	$\bot$	
$\oslash$	$\oplus$	δ	$=$	δ	
$\{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}$	$\oplus$	$\{f_1 \mapsto \delta'_1; \ldots; f_n \mapsto \delta'_n\}$	$=$	$\{f_1 \mapsto \delta_1 \oplus \delta'_1; \ldots; f_n \mapsto \delta_n \oplus \delta'_n\}$	
$\{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}$	$\oplus$	$\top$	$=$	$\{f_1 \mapsto \delta_1 \oplus \top; \ldots; f_n \mapsto \delta_n \oplus \top\}$	
$[C_1 \mapsto \delta_1; \ldots; C_n \mapsto \delta_n]$	$\oplus$	$[C_1 \mapsto \delta'_1; \ldots; C_n \mapsto \delta'_n]$	$=$	$[C_1 \mapsto \delta_1 \oplus \delta'_1; \ldots; C_n \mapsto \delta_n \oplus \delta'_n]$	
$[C_1 \mapsto \delta_1; \ldots; C_n \mapsto \delta_n]$	$\oplus$	$\top$	$=$	$[C_1 \mapsto \delta_1 \oplus \top; \ldots; C_n \mapsto \delta_n \oplus \top]$	
$\langle \delta_{def} \rangle$	$\oplus$	$\langle \delta'_{def} \rangle$	$=$	$\langle \delta_{def} \oplus \delta'_{def} \rangle$	
$\langle \delta_{def} \rangle$	$\oplus$	$\langle \delta'_{def} \triangleright i : \delta'_{exc} \rangle$	$=$	$\langle \delta_{def} \oplus \delta'_{def} \triangleright i : \delta_{def} \oplus \delta'_{exc} \rangle$	
$\langle \delta_{def} \triangleright i : \delta_{exc} \rangle$	$\oplus$	$\langle \delta'_{def} \triangleright j : \delta'_{exc} \rangle$	$=$	$\langle \delta_{def} \oplus \delta'_{def} \triangleright i : \delta_{def} \oplus \delta'_{exc} \rangle$ $\langle (\delta_{def} \vee \delta_{exc}) \oplus (\delta'_{def} \vee \delta'_{exc}) \rangle$	where $i = j$ otherwise
$\langle \delta_{def} \rangle$	$\oplus$	$\top$	$=$	$\langle \delta_{def} \oplus \top \rangle$	
$\langle \delta_{def} \triangleright i : \delta_{exc} \rangle$	$\oplus$	$\top$	$=$	$\langle \delta_{def} \oplus \top \triangleright i : \delta_{exc} \oplus \top \rangle$	
$\top$	$\oplus$	$\top$	$=$	$\top$	

TABLE 5.3 – $\oplus$ – Reduction Operator

Finally, the extractions summarized in Table 5.4, have been defined for dependencies δ and are used to express the data-flow equations of Section 5.3.

Definition 5.2.5. Extraction of a field's dependency.

$$.f \quad : \quad \mathcal{D} \nrightarrow \mathcal{D}.$$

Definition 5.2.6. Extraction of a constructor's dependency.

$$@C \quad : \quad \mathcal{D} \nrightarrow \mathcal{D}.$$

Definition 5.2.7. Extraction of an array's cell dependency.

$$\langle i \rangle \quad : \quad \mathcal{D} \nrightarrow \mathcal{D}.$$

Definition 5.2.8. Extraction of an array's dependency outside a cell i.

$$\langle * \setminus i \rangle \quad : \quad \mathcal{D} \nrightarrow \mathcal{D}.$$

Definition 5.2.9. Extraction of an array's general dependency.

$$\langle * \rangle \quad : \quad \mathcal{D} \nrightarrow \mathcal{D}.$$

They are partial functions, and can only be applied on dependencies of the corresponding kind. For instance, the field extraction $.f$ only makes sense for atomic or structured values with a field named f, which should be the case if the dependency represents a variable of a structured type with some field f. For any of the atomic dependencies δ_a, applying any of the defined extractions yields δ_a.

Table 5.4 – Dependency Extractions

$\delta.f, f \in \mathcal{F}$
$\top.f = \top$
$\oslash.f = \oslash$
$\bot.f = \bot$
$\{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}.f = \delta_i$ if $f = f_i$

$\delta@C, C \in \mathcal{C}$
$\top@C = \top$
$\oslash@C = \oslash$
$\bot@C = \bot$
$[C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m]@C = \delta_j$ if $C = C_j$

$\delta\langle * \setminus i \rangle$	$\delta\langle i \rangle$	$\delta\langle * \rangle$
$\top\langle * \setminus i \rangle = \top$	$\top\langle i \rangle = \top$	$\top\langle * \rangle = \top$
$\oslash\langle * \setminus i \rangle = \top$	$\oslash\langle i \rangle = \oslash$	$\oslash\langle * \rangle = \oslash$
$\bot\langle * \setminus i \rangle = \bot$	$\bot\langle i \rangle = \bot$	$\bot\langle * \rangle = \bot$
$\langle \delta_{def} \rangle \langle * \setminus i \rangle = \delta_{def}$	$\langle \delta_{def} \rangle \langle i \rangle = \delta_{def}$	$\langle \delta_{def} \rangle \langle * \rangle = \delta_{def}$
$\langle \delta_{def} \triangleright k : \delta_{exc} \rangle \langle * \setminus i \rangle = \begin{cases} \delta_{def} & \text{when } i = k \\ \delta_{def} \vee \delta_{exc} & \text{otherwise} \end{cases}$	$\langle \delta_{def} \triangleright k : \delta_{exc} \rangle \langle i \rangle = \begin{cases} \delta_{exc} & \text{when } i = k \\ \delta_{def} \vee \delta_{exc} & \text{otherwise} \end{cases}$	$\langle \delta_{def} \triangleright k : \delta_{exc} \rangle \langle * \rangle = \delta_{def} \vee \delta_{exc}$

5.2.2 Well-Typed Dependencies

The described syntactic dependencies are untyped. However, their interpretation is made in the context of a type τ. Dependencies such as $\oslash$ or $\top$ do not exhibit any data type features and can apply to any type, but others will be completely constrained, and most will fall in between, uncovering a few layers of structured types before reaching one of the "generic" leaves, $\oslash$, $\top$ or $\bot$. For example, the dependency $\{f \mapsto \delta_f\}$ only really makes sense for structured types with a single field f, whose type itself is compatible with δ_f, and shall not be used in connection with variant or array types.

As a consequence, we conclude the presentation of our abstract dependency type by explaining what it means for a dependency to be compatible with some type τ, i.e.

to be *well-typed* of some type τ. This is described as a judgement parameterized by the typing environment Γ (Definition 4.3.1) and the different inference rules are detailed in Table 5.5.

$$\frac{}{\Gamma \vdash \top \ : \ \tau}\ \text{WT}\top \qquad \frac{}{\Gamma \vdash \bot \ : \ \tau}\ \text{WT}\bot \qquad \frac{}{\Gamma \vdash \oslash \ : \ \tau}\ \text{WT}\oslash$$

$$\frac{\begin{array}{c}\tau = \boldsymbol{struct}\{f_1 : \tau_1, \ldots, f_n : \tau_n\} \\ \Gamma \vdash \delta_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash \delta_n : \tau_n\end{array}}{\Gamma \vdash \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} \ : \ \tau}\ \text{WTStruct}$$

$$\frac{\begin{array}{c}\tau = \boldsymbol{variant}[C_1 : \tau_1 | \ \ldots | \ C_n : \tau_n] \\ \Gamma \vdash \delta_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash \delta_n : \tau_n\end{array}}{\Gamma \vdash [C_1 \mapsto \delta_1; \ldots; C_n \mapsto \delta_n] \ : \ \tau}\ \text{WTVar}$$

$$\frac{\Gamma \vdash \delta_{def} \ : \ \tau}{\Gamma \vdash \langle \delta_{def} \rangle \ : \ \boldsymbol{arr}^{\tau_i}\langle \tau \rangle}\ \text{WTArr}$$

$$\frac{\Gamma \vdash \delta_{def} \ : \ \tau \qquad \Gamma \vdash \delta_{exc} \ : \ \tau \qquad \Gamma(i) = \tau_i}{\Gamma \vdash \langle \delta_{def} \triangleright i : \ \delta_{exc} \rangle \ : \ \boldsymbol{arr}^{\tau_i}\langle \tau \rangle}\ \text{WTArrI}$$

Table 5.5 – Well-Typed Dependencies

The atomic dependency values are *generic*: they are well-typed with respect to any type (WT$\top$, WT$\oslash$, WT$\bot$). The dependency δ for a structure (WTStruct) is well-typed only with respect to an adequate structured type, whose field types are themselves compatible with the dependency mapped to them in δ. Similarly, the dependency δ for a variant (WTVar) is well-typed only with respect to an adequate variant type. In turn, its constructors must be themselves compatible with the dependency mapped to them in δ. For well-typed array dependencies (WTArr, WTArrI), the default dependency as well as the exceptional dependency have to be compatible with the type τ of the array's elements. Furthermore, the type of i, the index of the known exceptional dependency has to be compatible with τ_i, the array's index type.

In the following section, we are discussing our intraprocedural dependency domain and the manner in which dependencies are computed and manipulated.

5.3 Intraprocedural Analysis and Data-Flow Equations

5.3.1 Intraprocedural Dependency Domains

At an intraprocedural level, dependency information has to be kept at each point of the control flow graph, for each variable of the typing environment Γ, that maps input,

output and local variables to their types. We use the term *domain* to denote this information.

Definition 5.3.1. Intraprocedural Dependency Domain $\Delta \in \mathscr{D}$. An *intraprocedural* domain $\Delta \in \mathscr{D}$:

$$\Delta : \mathcal{V} \to \mathcal{D}$$

is a mapping from variables to dependencies.

An intraprocedural domain is associated to every node of the control flow graph, representing the dependencies at the node's entry point. A special case is the mapping which binds all variables to $\bot$, which we call *Unreachable*:

$$\mathit{Unreachable} \equiv x \mapsto \bot.$$

In particular it is associated to nodes that cannot be reached during the analysis. Also, if any of the variables of Δ is marked as $\bot$, the entire node collapses, becoming *Unreachable*.

For any node of the control flow graph associated to an intraprocedural domain Δ, $\Delta(x)$ retrieves the dependency associated to the variable x. If a dependency for x has not been computed yet, it is mapped to $\oslash$.

Forgetting a variable x from a *reachable* intraprocedural domain, denoted by $\Delta \setminus x$, "erases" the variable's dependency information, by mapping it to $\oslash$.

Definition 5.3.2. Forget x.

$$\Delta \setminus x = \begin{cases} \mathit{Unreachable} & \text{when } \Delta = \mathit{Unreachable} \\ \Delta' = & y \mapsto \begin{cases} \Delta(y) & \text{when } y \neq x \\ \oslash & \text{when } y = x \end{cases} \end{cases}$$

The $\sqsubseteq_\Delta$, $\vee_\Delta$ and $\oplus_\Delta$ operations are pointwise extensions of $\sqsubseteq$ (defined in 5.2.2), $\vee$ (defined in 5.2.3), and $\oplus$ (defined in 5.2.4), respectively; they apply to intraprocedural dependency domains, for each variable and its associated dependency δ_v.

We define a partial order $\sqsubseteq_\Delta$ on $\mathscr{D}$.

Definition 5.3.3. Intraprocedural Partial Order $\sqsubseteq_\Delta$.

$$\sqsubseteq_\Delta \subseteq \mathscr{D} \times \mathscr{D}, \quad \Delta' \sqsubseteq_\Delta \Delta'' \ \textit{iff} \ \Delta'(x) \sqsubseteq \Delta''(x), \forall x \in \mathcal{V}.$$

In particular, *Unreachable* is the bottom of this intraprocedural lattice. It is the *identity* element of the intraprocedural join $\vee_\Delta$ operation and the absorbing element of the intraprocedural reduction operator $\oplus_\Delta$ defined below.

Definition 5.3.4. Intraprocedural Join Operation $\vee_\Delta$.

$$\vee_\Delta : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$$

$$\Delta' \vee_\Delta \Delta'' = \Delta \iff \Delta(x) = \Delta'(x) \vee \Delta''(x), \forall x \in \mathcal{V}.$$

Definition 5.3.5. Intraprocedural Reduction Operator $\oplus_\Delta$.

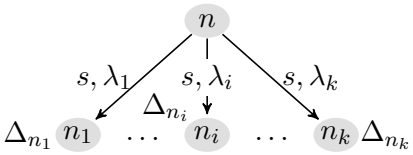
$$\oplus_\Delta : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$$

$$\Delta' \oplus_\Delta \Delta'' = \Delta \iff \Delta(x) = \Delta'(x) \oplus \Delta''(x), \forall x \in \Gamma.$$

Finally, an intraprocedural domain Δ is *well-typed* with respect to a typing environment Γ if and only if the dependency mapped to any variable x is well-typed with respect to x's type in the typing environment Γ (Definition 4.3.1).

5.3.2 Intraprocedural Data-Flow Equations

TABLE 5.6 – Statements – Representations and Data-Flow Equations

Representation	Equation
	$\Delta_n = \bigvee_\Delta \underset{n \xrightarrow{s,\lambda_i} n_i}{} [\![s]\!]_{\lambda_i}(\Delta_{n_i})$

Our dependency analysis is a *backward* data-flow analysis. For each exit label, it traverses the control flow graph starting with its corresponding exit node and it marks all other exit points as *Unreachable*, since exit labels are mutually exclusive. The intraprocedural domain for the currently analysed label is initialized with its associated output variables mapped to $\top$. Thereby, the analysis starts by making a conservative approximation and by considering that all the input has been observed and the output depends on it entirely. Typically, dependence analyses are *forward* analyses. However, given our goal to express *label-specific* dependencies as input-output relations and taking into consideration the characteristics of the αSmil language, choosing to design our analysis as a backward data-flow analysis seemed a pertinent choice. In αSmil, outputs are associated to a particular exit label and they are generated if and only if the predicate exits with that particular label. By traversing the control flow graph backwards, we can use this information and consider, starting with the initialisation phase, only the outputs that are relevant for the analysed exit label.

After the initialisation, the analysis then traverses the control flow graph and gradually refines the dependencies until a fixed point is reached. Table 5.6 summarizes the representation and general equation of the statements. For each statement, the presented data-flow equation operates on the intraprocedural domains of the statement's *successor* nodes. The intraprocedural domain at the *entry point* of the node is obtained by *joining* the contributions of each *outgoing* edge as shown in Figure 5.10.

Definition 5.3.6. The contribution of an edge (n_i, n_j) labeled with s and λ is given by $[\![s]\!]_\lambda(\Delta_{n_j})$ where $[\![s]\!]_\lambda(.)$ is the *transfer function* of the edge labeled s, λ.

Dependencies corresponding to variables that are written by a statement s on an exit label λ, denoted by $\mathsf{gen}_{s,\lambda}$ in Figure 5.10, are forgotten from the intraprocedural domain on which we are operating.

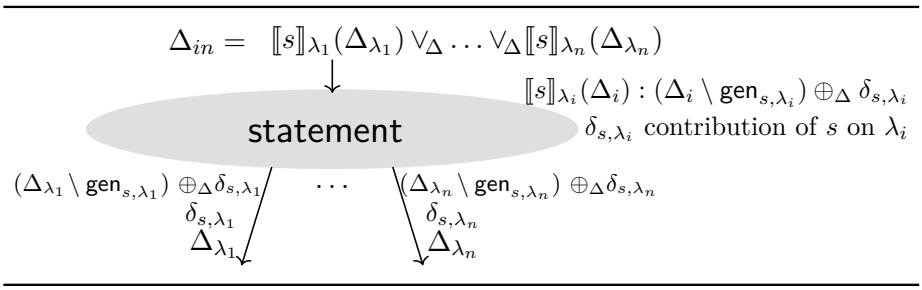


FIGURE 5.10 – Computation of the Intraprocedural Domain at a Node's Entry Point

In Section 5.2.1 we explained that as a consequence of the *non-monotonic* approximations made when joining dependencies corresponding to arrays, the result of the join operation is an upper bound, not a *least upper bound*. In order to deal with this issue, we adopt the generic solution consisting of systematically joining the dependency domain associated to a node before its iteration with the new dependency domain computed by the transfer function. Thus, the dependency domain of a node n is:

$$\Delta_n = old(\Delta_n) \vee_\Delta \big(\bigvee_{n \rightarrow n'}^{\Delta} [\![s]\!]_\lambda(\Delta_{n'}) \big).$$

This is not prohibitive in terms of performance, leading to an increase of the execution time of 5% to 10%.

Tables 5.7, 5.8, 5.9, 5.10 define the transfer functions for each built-in statement of our language, whereas the general case of a predicate call and its corresponding equation will be detailed in Section 5.4.

Table 5.7 presents the transfer functions for statements which are not type-specific. For equality tests (1) both of the inputs e_1, e_2 are completely read, whether the test returns true or false. The transfer functions therefore, reduce the domain of the corresponding successor node with a domain consisting of e_1 and e_2 both mapped to $\top$. In the case of assignment (2), the dependency of the written output variable o is forgotten from the successor's intraprocedural domain, thus being mapped to $\oslash$ and forwarded to the input variable e. The transfer function for the ***nop*** operation (3) is simply the identity.

Statement		$[\![s]\!]_{\lambda_i}(\Delta)$	
Equality test	(1)	$[\![e_1 = e_2]\!]_{\mathsf{true}}(\Delta) = \Delta \ \oplus_\Delta \ dep$ $[\![e_1 = e_2]\!]_{\mathsf{false}}(\Delta) = \Delta \ \oplus_\Delta \ dep$	where $dep = \left\{ \begin{array}{ll} e_1 & \mapsto \top \\ e_2 & \mapsto \top \end{array} \right\}$
Assignment	(2)	$[\![o := e]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus o) \ \oplus_\Delta \ \{e \mapsto \Delta(o)\}$	
No Operation	(3)	$[\![\boldsymbol{nop}]\!]_{\mathsf{true}}(\Delta) = \Delta$	

TABLE 5.7 – Generic Statements – Data-Flow Equations

The data-flow equations given in Table 5.8 correspond to structure-related statements. For the equations (4), (5), (6) and (7) we assume that the variable r is of type $\boldsymbol{struct}\{f_1 : \tau, \ldots, f_n : \tau\}$ for some fields f_i, $1 \leq i \leq n$. The equation (4) refers to the creation of a structure: each input e_i is read as much as the corresponding field f_i of the structure is read. The destructuring of a structure is handled in (5): each field f_i is needed as much as the corresponding variable o_i is. When accessing the i-th field of a structure r (6), only the field f_i is read, and only as much as the access' result o itself. The equation (7) treats field updates: the variable e_i is read as much as the field f_i is. The structure r is read as much as all the fields other than f_i are read in r'. Finally, the equations given in (8) handle partial structure equality tests, and the transfer functions are the same for the labels true or false: for both compared structures r' and r'', all the fields in the given set $f_1, \ldots, f_k$ are completely read, and only those.

Statement		$[\![s]\!]_{\lambda_i}(\Delta)$	
Create	(4)	$[\![r := \{e_1, \ldots, e_n\}]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus r) \ \oplus_\Delta \ \bigoplus_{1 \leq i \leq n} \{e_i \mapsto \Delta(r).f_i\}$	
Destructure	(5)	$[\![\{o_1, \ldots, o_n\} := r]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus \{o_i \mid o_i \in \bar{o}\}) \ \oplus_\Delta \ \{r \mapsto \{f_1 \mapsto \Delta(o_1); \ldots; f_n \mapsto \Delta(o_n)\}\}$	
Access field	(6)	$[\![o := r.f_i]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus o) \ \oplus_\Delta \ \{r \mapsto \{f_1 \mapsto \oslash; \ldots; f_i \mapsto \Delta(o); \ldots; f_n \mapsto \oslash\}\}$	
Update field	(7)	$[\![r' := \{r \ \boldsymbol{with} \ f_i = e\}]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus r') \ \oplus_\Delta \ \left\{ \begin{array}{lll} e_i & \mapsto & \Delta(r').f_i \\ r & \mapsto & \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} \end{array} \right\}$ where $\delta_j = \left\{ \begin{array}{ll} \Delta(r').f_j & \text{if } j \neq i \\ \oslash & \text{otherwise} \end{array} \right.$	
Equality	(8)	$[\![r' = \langle f_1, \ldots, f_k \rangle r'']\!]_{\mathsf{true}}(\Delta) = \Delta \ \oplus_\Delta \ d$ $[\![r' = \langle f_1, \ldots, f_k \rangle r'']\!]_{\mathsf{false}}(\Delta) = \Delta \ \oplus_\Delta \ d$	where $d = \left\{ \begin{array}{lll} r' & \mapsto & \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} \\ r'' & \mapsto & \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} \end{array} \right\}$ and $\delta_i = \left\{ \begin{array}{ll} \top & \text{if } f_i \in \{f_1, \ldots, f_k\} \\ \oslash & \text{otherwise} \end{array} \right.$

TABLE 5.8 – Structure-Related Statements – Data-Flow Equations

The data-flow equations given in Table 5.9 correspond to variant-related statements. They follow the same principles as those used for structure-related statements above. Note that the transfer functions for the switch (10) and possible constructor test (11) introduce $\bot$ dependencies for constructors which are known to be impossible on the considered edge. In particular, since $\bot$ is an absorbing element for $\oplus$, these transfer functions erase, for every constructor which is known to be locally impossible, all the dependency information possibly attached to such a constructor in the successor nodes. This is the actual *raison d'être* for the reduction operator, since using $\vee_\Delta$ to combine a successor domain and a local contribution would lose this information.

Finally, the equations for array-related statements are given in Table 5.10. We assume for both that the context is fixed and that $\mathcal{I}$ is the distinguished set of input variables for the analysed predicate. This set is used to make sure that exceptions in array dependencies are only registered to variables in $\mathcal{I}$ and not local or output variables. The reason for such a constraint is pragmatic: input variables are not assignable in our language, and therefore they always represent the same value intraprocedurally. Otherwise, each time a variable is written by a statement, we would need to traverse all the dependencies in the domain to erase or reinterpret the occurrences where this variable appears as an exception. Only recording exceptions for input variables makes this kind of costly traversal useless, and since only exceptions about input variables make sense at the interprocedural level (see Section 5.4), we do not lose much precision by doing so.

Statement		$[\![s]\!]_{\lambda_i}(\Delta)$
Create variant	(9)	$[\![v := C_p[e]]\!]_{\mathsf{true}}(\Delta) = (\Delta \setminus v) \ \oplus_\Delta \ \{e \mapsto \Delta(v)@C_p\}$
Variant Switch	(10)	$[\![\textbf{\textit{switch}}(v)\ \textbf{\textit{as}}\ [o_1 \mid \ldots \mid o_n]]\!]_{\lambda_i}(\Delta) = (\Delta \setminus o_i) \oplus_\Delta \{v \mapsto dep_i\}$ where $dep_i = [C_1 \mapsto \bot; \ldots; C_i \mapsto \Delta(o_i); \ldots; C_n \mapsto \bot]$
		$[\![v \in \{C_1, \ldots, C_k\}]\!]_{\mathsf{true}}(\Delta) = \Delta \ \oplus_\Delta \ \{v \mapsto [C_1 \mapsto \delta_1; \ldots; C_n \mapsto \delta_n;]\}$
		where $\delta_i = \begin{cases} \Delta(v)@C_i & \text{if } C_i \in \{C_1, \ldots, C_k\} \\ \bot & \text{otherwise} \end{cases}$
Possible variant	(11)	$[\![v \in \{C_1, \ldots, C_k\}]\!]_{\mathsf{false}}(\Delta) = \Delta \ \oplus_\Delta \ \left\{v \mapsto \left[\bar{C}_1 \mapsto \bar{\delta}_1; \ldots; \bar{C}_n \mapsto \bar{\delta}_n;\right]\right\}$
		where $\bar{\delta}_i = \begin{cases} \Delta(v)@\bar{C}_i & \text{if } \bar{C}_i \notin \{C_1, \ldots, C_k\} \\ \bot & \text{otherwise} \end{cases}$

TABLE 5.9 – Variant-Related Statements – Data-Flow Equations

Statement		$[\![s]\!]_{\lambda_i}(\Delta)$
Array access	(12)	$[\![o := a[i]]\!]_{\mathsf{true}}(\Delta) = \begin{cases} (\Delta \setminus o) \oplus_\Delta \left\{ \begin{array}{lcl} i & \mapsto & \top \\ a & \mapsto & \langle \oslash \triangleright i : \Delta(o) \rangle \end{array} \right\} & \text{when } i \in \mathcal{I} \\ (\Delta \setminus o) \oplus_\Delta \left\{ \begin{array}{lcl} i & \mapsto & \top \\ a & \mapsto & \langle \Delta(o) \vee \oslash \rangle \end{array} \right\} & \text{when } i \notin \mathcal{I} \end{cases}$ $[\![o := a[i]]\!]_{\mathsf{false}}(\Delta) = \Delta \oplus_\Delta \left\{ \begin{array}{lcl} i & \mapsto & \top \\ a & \mapsto & \langle \oslash \rangle \end{array} \right\}$
Array update	(13)	$[\![a' := [a \textbf{ with } i = e]]\!]_{\mathsf{true}}(\Delta) = \begin{cases} (\Delta \setminus a') \oplus_\Delta \left\{ \begin{array}{lcl} i & \mapsto & \top \\ e & \mapsto & \Delta(a')\langle i \rangle \\ a & \mapsto & \langle \Delta(a')\langle * \setminus i \rangle \triangleright i : \oslash \rangle \end{array} \right\} & \text{when } i \in \mathcal{I} \\ (\Delta \setminus a') \oplus_\Delta \left\{ \begin{array}{lcl} i & \mapsto & \top \\ e & \mapsto & \Delta(a')\langle * \rangle \\ a & \mapsto & \langle \Delta(a')\langle * \rangle \vee \oslash \rangle \end{array} \right\} & \text{when } i \notin \mathcal{I} \end{cases}$ $[\![a' := [a \textbf{ with } i = e]]\!]_{\mathsf{false}}(\Delta) = \Delta \oplus_\Delta \left\{ \begin{array}{lcl} i & \mapsto & \top \\ a & \mapsto & \langle \oslash \rangle \end{array} \right\}$

TABLE 5.10 – Array-Related Statements – Data-Flow Equations

The transfer functions for (12) and (13) thus take care of making adequate approximations when exceptions cannot be introduced. As for the cases when the array access exits with the false label, note that the contribution to the array a is $\langle \oslash \rangle$, which is strictly less precise than $\oslash$. The operation makes implicit bounds checking and this can thus be seen as accounting for the fact that no cell in a has been read, but the "length" or "support" of a has been read. Hence it would not be correct to claim that the result of the statement does not depend on a at all. Similarly, a variant dependency $[C_1 \mapsto \oslash, \ldots, C_n \mapsto \oslash]$ mapping all constructors to nothing has not read any value in any of the constructors, but may still depend on the variant's constructor itself. In contrast, we do not make this distinction for structures because we assume *surjective pairing*, i.e. structure values consist only of the fields themselves. Our solution can easily be adapted in order to deal with non-surjective cases.

5.3.3 Intraprocedural Dependency Analysis Illustrated

To better illustrate our analysis at an intraprocedural level, we exemplify the mechanism behind it, step by step, on the predicate `thread`, discussed in Section 5.1.1. We consider the **true** execution scenario, apply our dependency analysis and compare the actual obtained results with the targeted ones depicted in Figure 5.5.

Since a predicate can only exit with one label at a time and we are considering the **true** label, we can map the nodes `None` and `oob` to *Unreachable*, as shown in Figure 5.11. This is an advantage of backwards analyses. For **true**, we make a pessimistic assumption and map the output `ti` to $\top$, considering that control on the output is external and

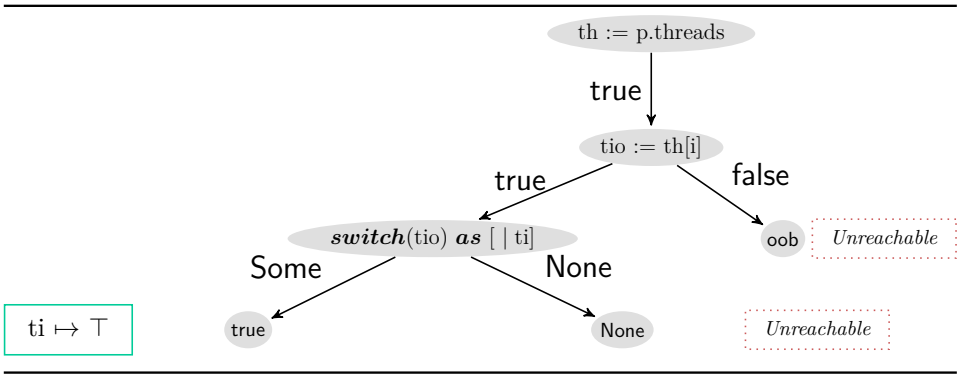
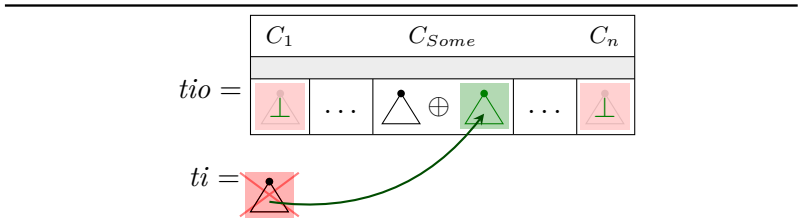


FIGURE 5.11 – Analysing Predicate `thread` – Initialisation

hence, out of our reach, and that `ti` will be entirely needed by a potential caller. Going further up the control flow graph, we analyse the *variant switch*.

In order to compute the dependency for the node corresponding to the variant switch, we apply the data-flow equation, given by (10) in Table 5.9. Since we are analysing the **true** case, we know that all other constructors (only the constructor `None` in this case) are locally impossible. Thus, we map it to $\bot$. We continue by forgetting the dependency information we knew about the output `ti`. Since its value is needed only in as much as the result of the switch on the corresponding edge is needed, we forward it to the part corresponding to the `Some` constructor. This is summarized below:



$$[\![\textbf{\textit{switch}}(v)\ \textbf{\textit{as}}\ [o_1|\ldots|o_n]]\!]_{\lambda_i}(\Delta) = (\Delta \setminus o_i) \oplus \{v \mapsto dep_i\}$$
where
$$dep_i = [\,C_1 \mapsto \bot;\ \ldots;\ C_i \mapsto \Delta(o_i);\ \ldots;\ C_n \mapsto \bot\,]$$

FIGURE 5.12 – Applying the Variant Switch Equation

Taking all this into account, for the node corresponding to the variant switch, we obtain the dependency shown in Figure 5.13. For the output `ti`, we depend entirely on the `Some` constructor of the node's input variant `tio`, while the constructor `None` is impossible.

Making a step further up the graph, we access the cell `i` of the array `th` and apply the equation (12) given in Table 5.10. We begin by forgetting the dependency for the output `tio`, since this is written. Since we only access the element `i`, we map all other cells to *Nothing*, i.e. $\oslash$. To the dependency corresponding to the `i`-th cell, we forward

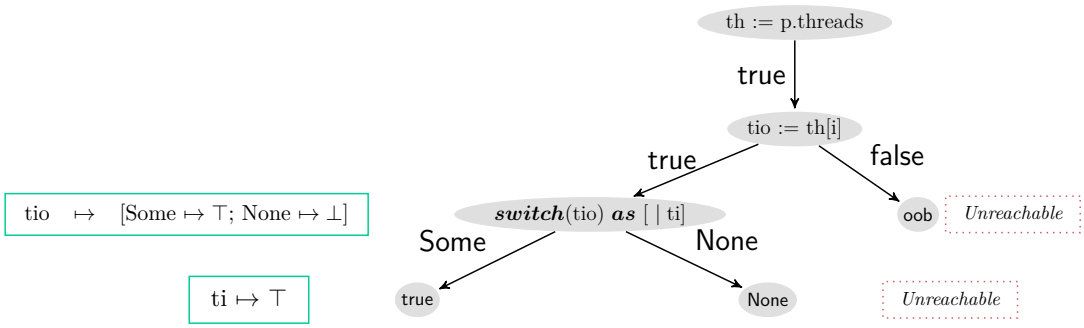


FIGURE 5.13 – Analysing Predicate `thread` – Variant Switch

the dependency we knew about `tio`, since we depend on it to the extent to which the result of the access is needed.

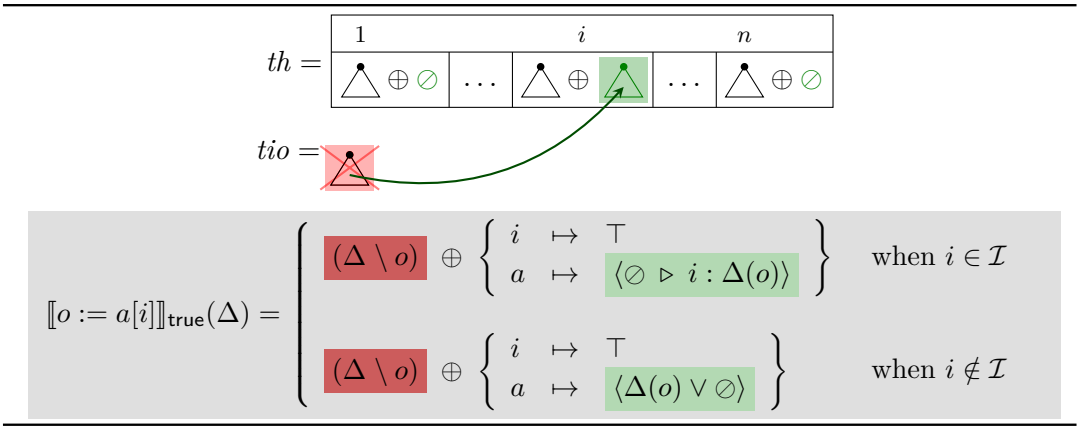


FIGURE 5.14 – Applying the Array Access Equation

We thus obtain a dependency stating that we depend only on the `i`-th cell of the array `th`, for which only the constructor `Some` is possible and entirely needed. The cell's index `i` is entirely needed as well. The applied equation is shown in Figure 5.14 (since `i` is an input, we use the first case of the equation) and the obtained results are shown in Figure 5.15.

As a last step, we access the field `threads` of the input process `p` and apply the equation (6) given in Table 5.8 and illustrated in Figure 5.16. As before, we forget the information for `th`, the access result. We map all other fields to ⊘ and we forward the dependency of the variable `th` to the dependency part of the field `threads`.

We thus obtain the dependency result shown in Figure 5.17. This states that for the label **true**, the output `ti` depends only on the `i`-th cell of the field `threads` of the input process `p`, for which it depends entirely on the `Some` constructor. Before returning the predicate's final results, the analysis filters out any dependency information referring to local variables and verifies that the invariant imposed on dependency information

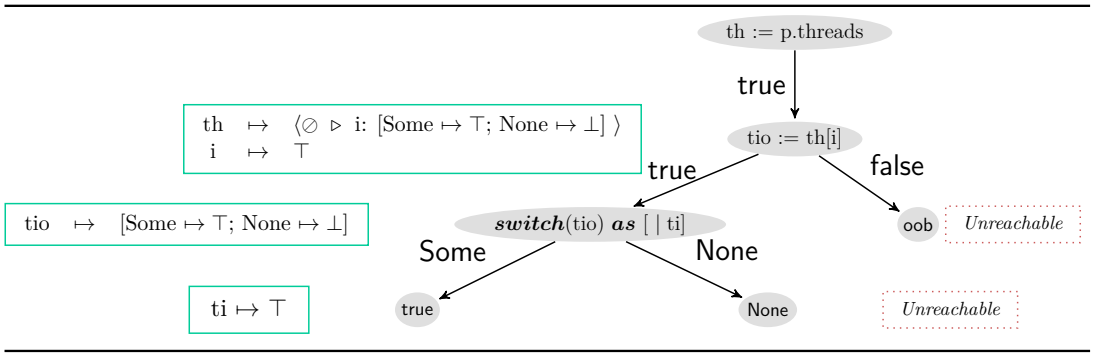


FIGURE 5.15 – Analysing Predicate `thread` – Array Access

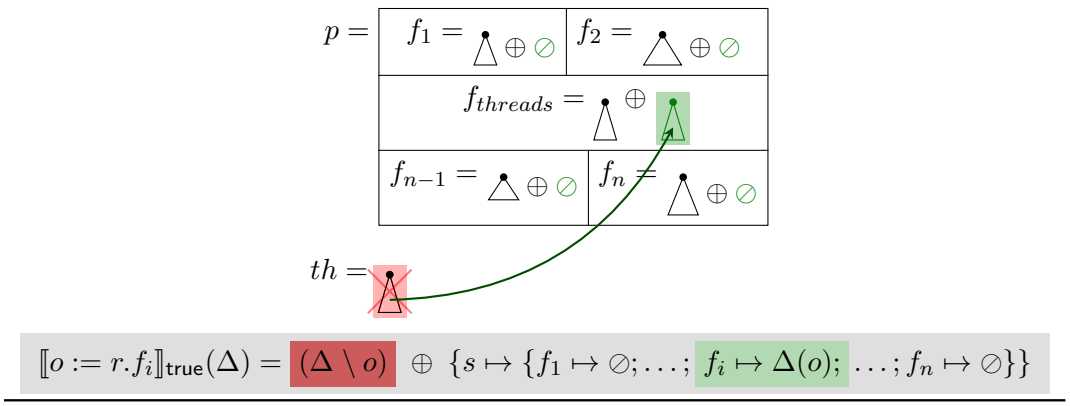


FIGURE 5.16 – Applying the Field Access Equation

related to arrays holds. Since the results refer only to the inputs `p` and `i` and the index of the exceptional computed dependency is an input, the invariant holds and the final result can be retrieved. The final dependency results obtained for the `thread` predicate on the exit label **true** are identical to the ones that we were targeting and that were depicted in Figure 5.5. For readability considerations, for structures such as the input process `p`, we omit dependencies on fields mapped to ⊘. We maintain this convention throughout the rest of this chapter, and thus any field of a structure that is omitted from a dependency summary should be interpreted as being mapped to ⊘, i.e. *nothing*.

5.4 Interprocedural Dependencies

Exit labels, presented in Section 3.1.2 and in Section 4.1 (on page 63), constitute an increased source of expressivity, as they indicate the scenario that was observed while executing a predicate. We incorporate this expressivity in our dependency results, by computing specific dependencies for each possible execution scenario. Therefore, our analysis is performed label by label and interprocedural dependency domains associate an intraprocedural domain to each exit label of the analysed predicate. The variable

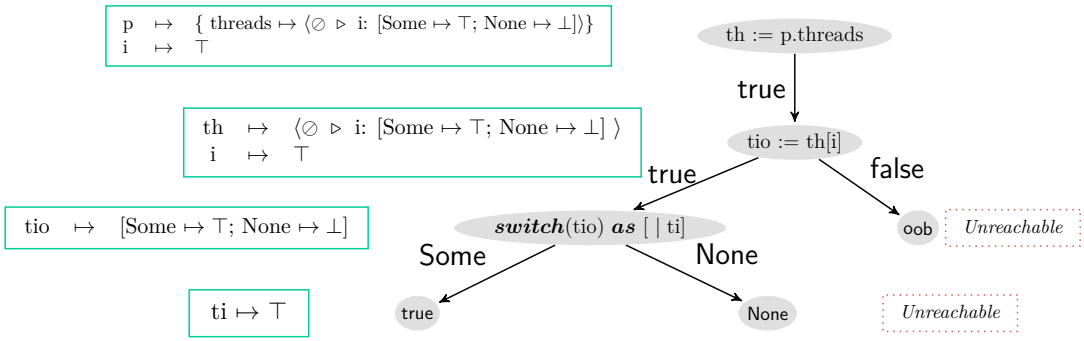


FIGURE 5.17 – Analysing Predicate `thread` – Field Access

key-set of each associated intraprocedural domain comprises the inputs of the analysed predicate. A label that cannot be returned is mapped to an *Unreachable* intraprocedural domain. This is a form of *path-sensitivity* (Robert and Leroy, 2012). However, we favor the term *label-sensitivity* for this characteristic, as it seems to be a more natural choice applied to our case and the language we are working on.

An interprocedural domain of a predicate p is thus defined as shown below.

Definition 5.4.1. Interprocedural Dependency Domain.

$$\mathscr{D}_p : \Lambda_p \to \mathscr{D}, \quad \text{where } \Lambda_p \text{ the set of output labels of predicate } p.$$

For each analysed label of a predicate, the analysis starts by initializing the intraprocedural domain mapped to it, with the output variables associated to the exit label. To avoid making any false assumption, these are initially mapped to the most general dependency, namely $\top$. Subsequently, as described in Section 5.3.2, the dependency information is gradually refined until a fixed point is reached. The execution scenarios denoted by the exit labels of a predicate are mutually exclusive. Therefore, during the analysis of a particular exit label, all other exit labels of the predicate are mapped to *Unreachable*. After reaching a fixed point, the intraprocedural domain is *filtered* so that only input variables appear in the variable set. As explained in Section 5.3.2, the intraprocedural domains are built such that only input variables may appear as exception indices in dependencies computed for arrays. This invariant is preserved throughout the analysis.

Interprocedural dependency information is expressed in terms of the formal parameters of predicates. For analysing predicate calls, we need to substitute the formal parameters of the callee, by the ones that are supplied by the caller. Therefore, a substitution must be performed on interprocedural summaries. This consists in substituting all occurrences of formal input parameters of a predicate by the corresponding effective input parameters. The substitution operation is denoted as $\blacktriangleleft(\chi)$, where χ is a substitution from formal to effective parameters.

We proceed by detailing the equation corresponding to a call to a predicate:

$$p(e_1, \ldots, e_n)[\lambda_1 : \bar{o}_1 \mid \ \ldots \ \mid \lambda_m : \bar{o}_m]$$

having the following signature:

$$p(\epsilon_1, \ldots, \epsilon_n)[\lambda_1 : \bar{\omega}_1 \mid \ \ldots \ \mid \lambda_m : \bar{\omega}_m].$$

The general equation (given in Table 5.6) applies:

$$\Delta_n = \bigvee_{\Delta \atop n \xrightarrow{s,\lambda_i} n_i} [\![p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]]\!]_{\lambda_i}(\Delta_{n_i}).$$

The transfer functions for the predicate call statement are deduced from the predicate's interprocedural domain in the following fashion:

$$[\![p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]]\!]_{\lambda_i}(\Delta) = (\Delta \setminus \bar{o}_i) \bigoplus_{j \in \{1,\ldots,n\}} e_j \mapsto dep^i_j$$

where (PredEq)

$$dep^i_j = \mathscr{D}_p(\lambda_i)(\epsilon_j) \blacktriangleleft (\bar{\epsilon} \mapsto \bar{e}).$$

Namely, the mappings for the outputs $\bar{o}$ associated to a label λ_i are removed, and the contribution of a call to each input e_j stems from the contribution of the interprocedural domain for label λ_i and formal input ϵ_j. In these, all the formal input parameters $\bar{\epsilon}$ in array dependency domains are substituted by the corresponding effective input parameters from $\bar{e}$.

An αSmil program is analysed by computing, once and for all an interprocedural dependency domain for every predicate. These are stored in a mapping binding predicate identifiers to their interprocedural dependency domains. Whenever a predicate call is handled intraprocedurally, the corresponding computed interprocedural dependency summary is retrieved from the mapping, propagated to the calling site and used as explained above. If an interprocedural dependency summary for a called predicate has not been computed yet, it is handled as if it were an *implicit* predicate. In practice, in programs generated in αSmil from Smil, predicates are sorted in topological order when possible. For *implicit* predicates described in Chapters 3 and 4, a pessimistic assumption is made: it is considered that everything in their inputs has been read and is needed, for any of their possible exit labels. Since their implementation is hidden, a conservative approximation must be made in their case.

Inductive predicates have been discussed in Section 3.1.4 (on page 46). They are *specification-only* predicates and represent a disjunction of cases. Each case can introduce existentially quantified variables. An inductive predicate exits with the true label if any of its declared cases holds. Therefore, for inductive predicates one analysis *per case* is made. For the true exit label, the dependency results are obtained by joining the results of all cases. For the false label, everything is considered to be read.

5.4.1 Interprocedural Dependency Analysis Illustrated

To better illustrate our analysis at an interprocedural level, we revisit our `start_address` example predicate introduced in Section 5.1.1. We consider the **true** execution scenario, apply our dependency analysis and compare the actual obtained results with the targeted ones depicted in Figure 5.8.

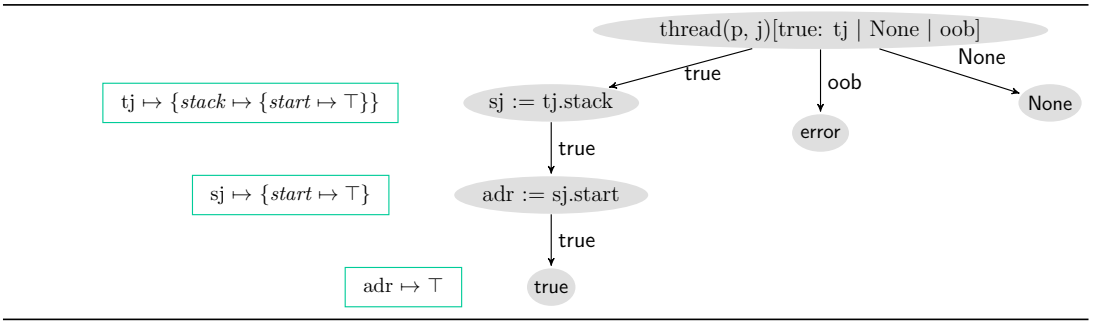


FIGURE 5.18 – $G_{start\_address}$ – Dependency Information

We begin by initialising the output `adr` with $\top$ and continue by traversing the control flow graph backwards and by computing the dependency information at each node. We apply the data-flow equation (6) given in Table 5.8 and we obtain the intermediate results shown in Figure 5.18.

To compute the dependency information of the control flow graph's entry node, i.e. the one corresponding to a predicate call to `thread`, we use the dependency summary computed for this predicate for the exit label **true** and we substitute the formal parameters, i.e. `p` and `i` appearing in it, with the effective arguments of the call, i.e. `p` and `j`. We thus obtain the following dependency summary:

$$\begin{array}{lcl} \mathrm{p} & \mapsto & \{\ \mathrm{threads} \mapsto \langle \oslash \ \triangleright\ \mathrm{j}\colon [\mathrm{Some} \mapsto \top;\ \mathrm{None} \mapsto \bot]\rangle\} \\ \mathrm{j} & \mapsto & \top \end{array}$$

We apply the data-flow equation (PredEq) corresponding to a predicate call, discussed on page 102, and make use of the dependency information corresponding to the successor node on the edge marked with **true**:

$$\mathrm{tj} \mapsto \{stack \mapsto \{start \mapsto \top\}\}$$

thus obtaining the following final dependency result:

$$\begin{array}{lcl} \mathrm{p} & \mapsto & \{\ \mathrm{threads} \mapsto \langle \oslash \ \triangleright\ \mathrm{j}\colon [\mathrm{Some} \mapsto \top;\ \mathrm{None} \mapsto \bot]\rangle\} \\ \mathrm{j} & \mapsto & \top \end{array}$$

However, the targeted results for `start_address` depicted in Figure 5.8 would translate to:

p ↦ { threads ↦ ⟨∅ ▷ j: [Some ↦ {t ↦ {*stack* ↦ {*start* ↦ ⊤}}}; None ↦ ⊥]⟩}
j ↦ ⊤

Clearly, the dependency information computed by our analysis and shown in Figure 5.19 is an over-approximation of the results that we had envisioned. The obtained dependency summary states that the entire `j`-th associated thread of the input process `p` is needed in order to obtain the output `adr` on the **true** exit label. However, in reality, only one of this thread's fields is actually needed, namely the `stack` field, for which only one subelement – the `start` field – is read. This loss of precision is a consequence of the dependency information mapped to the `Some` constructor at the control flow graph's entry node, corresponding to a call to the `thread` predicate. When executing successfully and exiting with label **true**, the `thread` predicate returns the `i`-th associated thread of its input process. However, the predicate `thread` does not need this element itself: it does not read nor use it *per se*, it merely retrieves it. The dependency on this returned element is relative to the amount in which the predicate's callers will use it. The `start_address` predicate for instance, depends only on one of the 3 fields of the returned thread. Yet, by mapping the `i`-th thread to ⊤ in `thread`'s dependency summary, we fail to mirror this distinction. ⊤ is the top element of our dependency domain and joining it with any other dependency will lead to ⊤, thus shadowing any other information we might compute while observing its usage.

5.4.2 Context-Insensitivity and its Consequences

Precision losses in dependency summaries, such as the one detected in our previous example, are a direct consequence of considering and analysing predicates in isolation. There is a level of information that goes beyond a predicate's own control flow graph and a more detailed picture that can emerge once non-local information connected to the predicate's use, i.e. the calling context, is included into the analysis.

Interprocedural analyses that consider the calling context when analysing the target of a function – or, in our case, a predicate – call are *context-sensitive* analyses (Hind, 2001). As the name implies, context-sensitive analyses can jump back to the original call site, using context information for the results they compute. *Context-insensitive* analyses on the other hand dispense with such information and propagate back to all

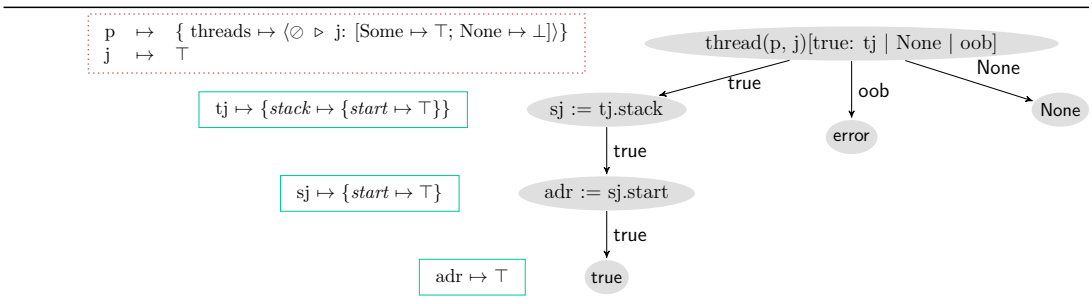


FIGURE 5.19 – $G_{start\_address}$ – Final Dependency Results

possible call sites the information that they compute once. This is a notorious source of potential precision loss in static analysis. Choosing either one of these two traits has significant consequences: on the one hand, by choosing to ignore the calling context and the additional information it supplies, one pays a high price in terms of precision, and on the other hand, by choosing to include such information, one risks sacrificing scalability.

Our dependency analysis as presented so far is context-insensitive: for each predicate, the analysis computes a dependency summary once, stores it, and further propagates it to its callers, whenever needed. Considering that αSmil predicates are sequences of calls to other predicates, built-in or user-defined, as discussed in Chapter 4, if we would adopt a purely context-sensitive solution, we would gain in terms of precision, but we would obtain results that are prohibitive in terms of performance. This is a typical trade-off of static analyses. We address this issue and describe our solution in detail in Chapter 6. Without adopting context-sensitivity to the letter, we strike a balance between the two alternatives by including lazy components in our interprocedural dependency summaries and by using them for injecting the current intraprocedural context on an as-needed basis. As will be discussed in Chapters 6 and 8, this approach leads to improved precision, with only a marginal decrease in performance.

5.5 Semantics of Dependency Values

There are two different manners of interpreting dependency values δ, one focusing on the possible constructors part and the other focusing on the dependency part. In both cases the interpretations are relative to a type τ and hold only for well-typed dependencies of the same type. The set of types that a dependency is compatible with has been discussed in Section 5.2.2 and defined in Table 5.5.

First, focusing on the possible constructors aspect, dependencies can be interpreted as a *constraint* on the forms that values may take. Such constraints can arise as a consequence of $\bot$, i.e. *impossible*, appearing in nested dependencies. These are described by a characteristic function $\mathbb{1}$:

$$\mathbb{D} = \{(v, \delta) \in \mathbb{D} \times \mathcal{D} \mid \ \delta \in \mathcal{D}, \tau \in \mathbb{T}, v \in \mathbb{D}_\tau, \Gamma \vdash \delta : \tau\}$$
$$\mathbb{1} : \mathbb{D} \to \{0, 1\}.$$

This is defined as follows below.

Definition 5.5.1. Characteristic function $\mathbb{1}$.

$$\mathbb{1}(v, \top) = 1$$
$$\mathbb{1}(v, \oslash) = 1$$
$$\mathbb{1}(v, \bot) = 0$$
$$\mathbb{1}(\{f_1 = v_1, \ldots, f_n = v_n\}, \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}) = \begin{cases} 1, & \text{when } \mathbb{1}(v_i, \delta_i), \forall 1 \leq i \leq n \\ 0, & \text{otherwise} \end{cases}$$

$$\mathbb{1}(C_i[v], [C_1 \mapsto \delta_1; \ldots; C_n \mapsto \delta_n]) = \begin{cases} 1, & \text{when } \mathbb{1}(v, \delta_i) \\ 0, & \text{otherwise} \end{cases}$$

$$\mathbb{1}((\mathcal{P}, (v_k)_{k\in\mathcal{P}}), \langle \delta_{def} \rangle) = \begin{cases} 1, & \text{when } \mathbb{1}(v_k, \delta_{def}), \forall k \in \mathcal{P} \\ 0, & \text{otherwise} \end{cases}$$

$$\mathbb{1}((\mathcal{P}, (v_k)_{k\in\mathcal{P}}), \langle \delta_{def} \triangleright i : \ \delta_{exc} \rangle) = \begin{cases} 1, & \text{when } (\mathbb{1}(v_k, \delta_{def}), \\ & \forall k \in \mathcal{P}, k \neq E(i)) \text{ or} \\ & (E(i) \in \mathcal{P}, \mathbb{1}(v_{E(i)}, \delta_{exc})) \\ 0, & \text{otherwise} \end{cases}$$

This interpretation is compatible with the partial order $\sqsubseteq$ (Definition 5.2.2, Table 5.1) defined on dependencies. If a dependency is more precise or equal to another dependency, then it should be interpreted as constraints which are at least as strong as the ones for the other dependency. Given a typing environment Γ (Definition 4.3.1):

$$\forall \tau \in \mathbb{T}^*, \delta \sqsubseteq \delta' \implies (\mathbb{D}_\tau \cap \mathbb{1}(\bullet, \delta)) \subseteq (\mathbb{D}_\tau \cap \mathbb{1}(\bullet, \delta'))$$

where

$$\mathbb{T}^* = \{\tau \in \mathbb{T} \mid \Gamma \vdash \delta : \tau \wedge \Gamma \vdash \delta' : \tau\}.$$

The interpretation of the reduction operator $\oplus$ (Definition 5.2.4) with respect to the constraints semantics of dependencies is that if two dependencies δ and δ' can be interpreted as constraints for a value v, then their reduction can be interpreted as a constraint for v as well:

$$\mathbb{1}(v, \delta) \wedge \mathbb{1}(v, \delta') \implies \mathbb{1}(v, \delta \oplus \delta').$$

The converse, which one might expect to be true as well, does not hold because of approximations made by our treatment of arrays.

Given a valuation E (Definition 4.4.2), an intraprocedural dependency summary can be interpreted as a conjunction of the constraints on every variable's value as given by its associated dependency. We use the notation $E \vDash \Delta$ to indicate this:

$$E \vDash \Delta \implies \forall v \in \mathcal{V}, \mathbb{1}(E(v), \Delta(v)).$$

Under the appropriate conditions, given a semantic transition $\xrightarrow{\lambda}$ (Definition 4.4.4) from the configuration $\langle E, [s] \rangle$ (Definition 4.4.3) to the valuation E', as defined in Section 4.4, if the intraprocedural summary Δ' of the statement's s successor on label λ represents the semantic interpretation of constraints given E', then the contribution $[\![s]\!]_\lambda(\Delta')$ (Definition 5.3.6) of the edge labeled with s and λ, must necessarily represent the semantic interpretation of constraints given E. We thus obtain the following:

$$\Gamma \vdash E \implies \tag{5.1}$$
$$\Sigma, \Gamma, \mathcal{O} \vdash s \to \lambda \implies \tag{5.2}$$
$$\langle E, [s] \rangle \xrightarrow{\lambda} E' \implies \tag{5.3}$$
$$\Gamma, E' \vdash \Delta' \implies \tag{5.4}$$
$$E' \vDash \Delta' \implies \tag{5.5}$$
$$E \vDash [\![s]\!]_\lambda(\Delta') \tag{5.6}$$

We note that thanks to the subject reduction property (Definition 4.4.7), (5.3) implies that $\Gamma \vdash E'$.

Following from (5.6), when joining the contributions on all labels of the statement s, the obtained intraprocedural dependency summary represents the semantic interpretation of the disjunction of constraints given E:

$$\begin{array}{l}(E \vDash [\![s]\!]_{\lambda_1}(\Delta'_1)) \vee_\Delta \ldots \vee_\Delta (E \vDash [\![s]\!]_{\lambda_n}(\Delta'_n)) \implies \\ E \vDash ([\![s]\!]_{\lambda_1}(\Delta'_1) \vee_\Delta \ldots \vee_\Delta [\![s]\!]_{\lambda_n}(\Delta'_n)) \implies \\ E \vDash \mathit{old}(\Delta) \implies \\ E \vDash \mathit{old}(\Delta) \vee_\Delta ([\![s]\!]_{\lambda_1}(\Delta'_1) \vee_\Delta \ldots \vee_\Delta [\![s]\!]_{\lambda_n}(\Delta'_n))\end{array}$$

For a predicate p exiting with label λ and having the intraprocedural summary Δ_λ, the characteristic function, given $\mathcal{I} \subseteq E$, a valuation mapping the predicate's inputs to their values, constrains the space of inputs that can make the predicate exit with the label λ. It thus denotes the necessary conditions on inputs according to the observed execution scenario and can be used as an *inversion lemma* when reasoning on calls to a predicate.

The *soundness* of this interpretation as well as the *well-formedness* of our dependencies have been proven in Coq and the corresponding files can be consulted online[1]. The mechanized Coq proofs are entirely due to Stéphane Lescuyer. These proofs also deal with *deferred dependencies* that will be presented in Chapter 6, but these constitute an extension that does not modify the underlying lattice.

The second interpretation of dependency values focuses on the dependency part and is a partial equivalence relation $\approx$:

$$\mathbb{TD} = \{(\tau, \delta) \in \mathbb{T} \times \mathbb{D} \mid \Gamma \vdash \delta : \tau\}$$
$$\approx : \mathbb{TD} \to \mathbb{D} \times \mathbb{D}.$$

The partial equivalence relation $\approx_\delta^\tau$ relates well-typed values of the same type τ. It relates values that only differ in places that are irrelevant according to the dependency δ. It is defined as shown below.

[1]The corresponding files are provided at the following address: http://ajl-demo.fr/2015/proveCoq/

Definition 5.5.2. Partial Equivalence Relation $\approx^{\tau}_{\delta}$.

$$
\begin{aligned}
\approx^{\tau}_{\top} &= \{(x,x) |\ x \in \mathbb{D}_{\tau}\} \\
\approx^{\tau}_{\oslash} &= \{(x,y) |\ x,y \in \mathbb{D}_{\tau}\} \\
\approx^{\tau}_{\bot} &= \{(x,y) |\ x,y \in \mathbb{D}_{\tau}\} \\
\approx^{\boldsymbol{struct}\{f_1:\tau_1,\ldots,f_n:\tau_n\}}_{\{f_1\mapsto\delta_1;\ldots;f_n\mapsto\delta_n\}} &= \{\ (\{f_1 = v_1,\ldots,f_n = v_n\},\{f_1 = w_1,\ldots,f_n = w_n\}) \mid \\
&\qquad \forall i, 1 \le i \le n,\ (v_i,w_i) \in \approx^{\tau_i}_{\delta_i}\} \\
\approx^{\boldsymbol{variant}[C_1:\tau_1 \mid \ldots \mid C_n:\tau_n]}_{[C_1\mapsto\delta_1;\ldots;C_n\mapsto\delta_n]} &= \{(C_i[v_i], C_i[w_i]) \mid\ (v_i,w_i) \in \approx^{\tau_i}_{\delta_i}\}
\end{aligned}
$$

$$
\approx^{\boldsymbol{arr}^{\tau_i}\langle\tau\rangle}_{\langle\delta_{def}\rangle} = \{((\mathcal{P},(v_k)_{k\in\mathcal{P}}),(\mathcal{P},(w_k)_{k\in\mathcal{P}})) \mid\ \forall k,(v_k,w_k) \in \approx^{\tau}_{\delta_{def}}\}
$$

$$
\begin{aligned}
\approx^{\boldsymbol{arr}^{\tau_i}\langle\tau\rangle}_{\langle\delta_{def}\triangleright i\,:\,\delta_{exc}\rangle} = \{\ &((\mathcal{P},(v_k)_{k\in\mathcal{P}}),(\mathcal{P},(w_k)_{k\in\mathcal{P}})) \mid\ E(i) \in \mathcal{P} \Longrightarrow \\
&(v_{E(i)},w_{E(i)}) \in \approx^{\tau}_{\delta_{exc}},\ \forall k \neq E(i),(v_k,w_k) \in \approx^{\tau}_{\delta_{def}}\}
\end{aligned}
$$

This interpretation is compatible with the partial order $\sqsubseteq$ (Definition 5.2.2) defined on dependencies. If a dependency is more precise or equal to another dependency, then it should be interpreted as an equivalence relation relating more values:

$$\delta \sqsubseteq \delta' \implies \approx^{\tau}_{\delta} \supseteq \approx^{\tau}_{\delta'},\ \ \forall\tau,\Gamma \vdash \delta : \tau \wedge \Gamma \vdash \delta' : \tau.$$

The interpretation of the reduction operator $\oplus$ (Definition 5.2.4) with respect to the equivalence relation interpretation of dependencies is that the set of values related by $\delta \oplus \delta'$ is a subset of the intersection of values related by δ and δ', respectively:

$$\approx^{\tau}_{\delta\oplus\delta'} \ \subseteq\ \approx^{\tau}_{\delta} \ \cap\ \approx^{\tau}_{\delta'},\ \ \forall\tau,\Gamma \vdash \delta : \tau \wedge \Gamma \vdash \delta' : \tau.$$

The interpretation of the $\vee$ operator (Definition 5.2.3, Table 5.2) with respect to the equivalence relation interpretation of dependencies is similar:

$$\approx^{\tau}_{\delta\vee\delta'} \ \subseteq\ \approx^{\tau}_{\delta} \ \cap\ \approx^{\tau}_{\delta'},\ \ \forall\tau,\Gamma \vdash \delta : \tau \wedge \Gamma \vdash \delta' : \tau.$$

Given two valuations E and E', they are equivalent modulo an intraprocedural dependency summary Δ if the values that they associate to variables are equivalent modulo the corresponding dependency associated in Δ:

$$E \approx^{\Gamma}_{\Delta} E' \implies \forall v \in \Delta, E(v) \approx^{\Gamma(v)}_{\Delta(v)} E'(v).$$

The equivalence relation $\approx^{\Gamma}_{\Delta}$ thus relates valuations that are not distinguishable by only looking at the parts specified by the intraprocedural dependency summary Δ.

This interpretation can be used to apply *congruence modulo* reasoning to predicate calls. By calling a predicate p with two sequences of input values $\bar{v}$ and $\bar{u}$, respectively,

which are related by the intraprocedural dependency summary of p on label λ, then the predicate will necessarily exercise the same execution scenario, exiting with label λ and will yield identical outputs $\bar{w}$.

5.6 Related Work

The *frame problem* and its manifestations in the software verification process – detecting program properties that remain unchanged under a certain operation – are notorious (Leavens, Leino, and Müller, 2007; Leavens and Clifton, 2005; O'Hearn, 2005). A *complete specification* of a program will necessarily include *frame properties* (Borgida, Mylopoulos, and Reiter, 1995). However, though necessary, specifying and verifying frame properties is tedious and repetitive. Two prominent solutions to the frame problem come from *separation logic* (Reynolds, 2005; Distefano, O'Hearn, and Yang, 2006; Calcagno et al., 2011) and *ownership types* (Clarke and Drossopoulou, 2002). However, Meyer (Meyer, 2015) argues that the problem itself should not impose such annotation-heavy solutions. Simpler, automatic solutions for their specification and verification would allow programmers to concentrate on the truly challenging part (Meyer, 2015).

Though we share the same desideratum with separation logic (Reynolds, 2002; Reynolds, 2005; O'Hearn, 2012; O'Hearn, Yang, and Reynolds, 2004), the programming paradigm and context under which we operate leads to a considerably different solution. Separation logic is targeted at low-level imperative programming languages and its applications focus on shared, mutable data structures. We, on the other hand, focus on a purely functional language and consider immutable algebraic data structures and arrays. We treat mappings between variables and values and analyse their evolution in a side-effect free environment, in the context of verification of programs where a new output is obtained by altering just a subset of the input's subelements and preserving the rest. Instead of using a collection of Hoare triples as an abstract domain, we have defined our own dependency domain. The results of our dependency analysis are close to the concept of a footprint (Distefano, O'Hearn, and Yang, 2006; Hur, Dreyer, and Vafeiadis, 2011; Bobot and Filliâtre, 2012), in the sense that they describe an over-approximation of only those variables and subelements that are needed by a program and are expressed as an input-output relation.

The dependency results computed by our analysis are similar to *primitive read* and *write effects* used in ownership type systems (Clarke and Drossopoulou, 2002). Write effects in our case are implicit and include strictly the output variables associated to an exit label. Read effects can only refer to input variables of a predicate. Also, read effects comprise the whole execution of a method even if they are irrelevant for the method's results. We however, ignore read effects on which the output does not depend, reflecting only those which contribute to the observed result. A technique for declaring and verifying read effects in an ownership type system is presented in (Clarke and Drossopoulou, 2002). We use static analysis to automatically detect them. In the Spec# (Mike Barnett, 2005) program verifier, the notion of *confined* is used for

describing the reading effects of a pure method in terms of the *ownership cone* (Clarke, Potter, and Noble, 1998) of its parameters.

In (Hughes, 1987), Hughes argues that analyses of programs that manipulate data structures, should ideally distinguish between the information they are computing for a data structure as a whole, and the information computed for each component within it. The information that is computed by a backward analysis is dubbed generically as *context*. A manner of constructing richer domains is described and it is argued that for instance, a context for a *sum type* must contain (sub)contexts for any of its *summands*. Similarly, for *product types*, a context should include a (sub)context for each component, as well as a context referring to the value as a whole. We target fine-grained dependency information for structures, variants and arrays. Similarly to the described product type contexts, our dependencies for structures describe the dependency on each of the structure's fields. Variant dependencies are expressed in terms of the dependencies of their constructors, i.e. their summands. Furthermore, it is argued that any context should include a maximal element, interpreted as a "no information" value, a minimal element, interpreted as "contradictory requirements" and an element representing "no context" or "unused". Close to the notion of "contradictory requirements", we include an atomic value denoting *impossible* in our dependency domain. Program points having a "contradictory requirements" context denote points in the program that will lead to crashes if reached. Our notion of *impossible* refers to nodes that are unreachable or constructors that cannot occur on a given execution path. Our maximal element, denoting *everything* is a safe value, close to the notion of "no information". *Nothing*, an element different from both *everything* and *impossible*, is similar to the notion of "unused". It denotes (sub)elements that are irrelevant and constitutes quite definite information.

Hughes (Hughes, 1987) introduces a notion of needed/unneeded parameters for programs manipulating lists. This enables detecting whether the value of a subterm is ignored. The method is formulated in terms of a fixed, finite set of projection functions. Multiple other approaches and analyses focus on the elimination of unnecessary data structures (Cousot and Cousot, 1994), filtering of useless arguments and unnecessary variables in the context of logic programming (Leuschel and Sørensen, 1996), and more recently, removing redundant arguments (Alpuente, Escobar, and Lucas, 2007).

The concept of a *context* is further discussed by Wadler and Hughes in (Wadler and Hughes, 1987). The authors describe a technique for strictness analysis for non-flat list domains that relies on contexts represented using the notion of *projections* from domain theory. These allow expressive list descriptions, such as contexts specifying that while a list's elements can be ignored, its length is relevant. Their backward analysis computes *necessary* information using a fixed finite abstract domain.

Leino and Müller (Leino and Müller, 2008b) present a technique for verifying that methods that query the state of identical data structures, return identical or equivalent results. They stress the frequency of such assumptions in program verification, as well as the counter-intuitive amount of effort required for the specification and verification of such *equivalent-results* methods and their callers. One of the two interpretations of our dependency values — $\approx^{\tau}_{\delta}$ — is an equivalence relation binding pairs of values

that are not distinguishable by considering only the parts specified by the dependency domain. Thus, it ensures not only that identical input data structures will lead to identical results but also that different invocations of a predicate with input data structures that are congruent with respect to this interpretation will lead to identical results. Our dependencies are similar to the *influence sets* presented by Leino and Müller. Influence sets are represented as sets of heap locations and they are used to specify the parts of the program state that are allowed to impact the return values. Influence sets are user-defined and they are required to be *self-protecting*. This property is enforced by requiring the set of path expressions specifying the influence set to be *prefix close*, a constraint which is then checked syntactically. In contrast, our dependencies are computed by static analysis. Influence sets may depend on the heap. Reasoning about heap locations is beyond the scope of our analysis. We treat mappings between variables and values, analyse their evolution in a side-effect free environment and express dependencies as input-output relations. The technique presented by Leino and Müller has been applied for reasoning about *pure methods* (Leino, Müller, and Wallenburg, 2008; Hatcliff et al., 2012; Nordio et al., 2010; Banerjee and Naumann, 2014).

Identifying the input (sub)parts on which a predicate's outputs depend can also be seen as an instance of *secure information flow* (Sabelfeld and Myers, 2003), where the predicate's outputs and the input (sub)parts appearing in the predicate's dependency summary have a low-security level, i.e. are public, and everything else has a high-security level, i.e. is private. The first interpretation of our dependency values mirrors the notion of *non-interference* as given by Volpano et. al. in (Volpano, Irvine, and Smith, 1996) for deterministic programs. By only observing the public parts, nothing can be concluded about the private parts. The link between permissions and ownership types has been underlined by Zhao and Boyland (Zhao and Boyland, 2008).

Liu and Stoller present a backward dependence analysis for the computation of dead code (Liu and Stoller, 2003). They obtain expressive descriptions of partially dead recursive data using *liveness patterns*. These are based on general regular tree grammars that were extended with two notions: *live* and *dead*. Users can specify liveness patterns at particular program points of interest. The analysis then uses these and computes liveness patterns at all program points based on constraints derived from the programming language semantics and the program itself. The obtained information is meant to be used for identifying and eliminating dead code. In a separate paper (Liu, 1998), Liu presents three approximation operations meant to guarantee termination in the context of fixed point computations using general grammar transformers on potentially infinite grammar domains.

Static dependence or liveness analyses are typically used for code optimization, dead code elimination (Liu and Stoller, 2003) and compile time garbage collection, but only seldom for program verification. One exception that we are aware of comes from Frama-C (Cuoq et al., 2012), where it is used in a purely *automatic* setting and unlike our analysis it does not handle unions and arrays. A plug-in based on the available value analysis (*Frama-C Value Analysis User Manual*) computes lists of input and output locations for each function, distinguishing between *operational*, *functional* and *imperative inputs and outputs*. Dependencies computed for an output o hold if

and when the analysed function terminates. They are represented as sets of variables, whose initial value can influence the final value of o. Input variables appearing in this set are called *functional inputs*. *Imperative inputs* are the locations that may be read during the execution of the analysed function. An over-approximation of the set of these locations is computed; locations that are read only in non-terminating branches are included in the imperative inputs set as well. *Operational inputs* are the memory zones that are read without having been previously written to.

5.7 Conclusion

In the context of interactive formal verification of complex systems, considerable effort is spent on proving the preservation of the system's invariants. However, most operations have a localised effect on the system, which only really impacts few invariants at the same time. Identifying those invariants that are unaffected by an operation can substantially ease the proof burden for the programmer.

In this chapter, we have presented a data-flow analysis that computes a conservative approximation of the input fragments on which the operations depend. It is a *flow-sensitive*, *path-sensitive*, interprocedural dependency analysis that handles arrays, structures and variants. For the latter it simultaneously computes a subset of possible constructors. We have defined our own abstract dependency domain and we obtain dependency information that mirrors the layered structure of compound data types.

The main original traits of this contribution stem from its design as an analysis meant to be used as a companion tool during interactive program verification, in a unified manner on programs as well as on specifications.

We have implemented a prototype of the dependency analysis in OCaml and we have applied it to a functional specification of ProvenCore (Lescuyer, 2015), a general-purpose microkernel that ensures *isolation*. Its proof is based on multiple refinements between successive models, from the most abstract one, on which the isolation property is defined and proven, to the most concrete, i.e. the actual model used for code generation. Medium-sized experiments performed on the abstract layers of ProvenCore show positive results. For instance, the dependency results of approximately 630 αSmil predicates, totalling approximately 10000 lines of code are obtained in less than 1 second. Static approaches have long been considered as being confined to small programs. We believe that our preliminary results indicate that it is possible to report conservative dependency summaries without sacrificing scalability. The implementation and the obtained results will be presented and discussed in detail in Chapter 8. The prototype can be tested on the web page[2] dedicated to our dependency analysis, where various examples are provided and explained. Additionally, users can devise and test their own examples.

An obvious first challenge is to address the issue of *context-sensitivity*. In the following chapter, we present a solution based on lazy components which are included in our interprocedural dependency summaries. The current intraprocedural context is

[2]Dependency Analysis Web Page: http://ajl-demo.fr/2015/

injected in them on an as-needed basis. As we will show in Chapter 6, these lead to improved precision, with only a marginal decrease in performance.

Our main goal is to combine the dependency analysis with the correlation analysis presented in Chapter 7, which is meant to detect relations between inputs and outputs. By uncovering partial equivalence relations between inputs and outputs, after having detected that a property only depends on unmodified parts and by unifying the results, the preservation of invariants for the unmodified parts can be inferred.

We surmise that besides its intended target, other programming activities can rely on our dependency analysis as well. For instance, it could have applications in the testing realm: the computed dependency information could be used for designing and generating test suites that avoid redundant testing of the same execution scenario. Based on the second interpretation — $\approx_{\delta}^{\tau}$ — of our dependency information, given in Section 5.5, classes of inputs that will test the same execution scenario can be determined. The input subelements on which the outputs of a predicate do not depend can be consistently supplied with the same testing value, as they are completely irrelevant for the outcome. On the contrary, the input subelements on which the outputs depend, should be targeted and their values should be varied for more comprehensive testing. Since our dependency analysis computes results for every exit label of an αSmil predicate, it could also facilitate unit testing for exceptions. Furthermore, the computed dependency information could provide assistance in specifying *read effects* of predicates, similar to *accesible clauses* (Leavens et al., 2006) in JML.

The dependency analysis presented in this chapter has been the subject of a previous publication (Andreescu, Jensen, and Lescuyer, 2015).

Chapter 6

Deferred Dependencies: Injecting Context in Dependency Summaries

> No symbols where none intended.
>
> Samuel Beckett

6.1 Dealing with Context-Insensitivity

Traditionally, the precision of static analyses is characterized along several axes, including the scope of the analysis, i.e. *intraprocedural* or *interprocedural* analyses, and different nuances of *sensitivity*, relative to the analysis' use of control-flow information or of information pertaining to the calling context. This classification and terminology has its origins in data-flow analyses (Hind, 2001; Midtgaard, 2012). Regarding scope, *intraprocedural* analyses are local and operate within the boundaries of procedures. In contrast, *interprocedural* analyses are global and operate across procedure calls (Midtgaard, 2012). These are somewhat more challenging and costly to perform and impose dealing with parameter mechanisms.

Another important distinction is made regarding the calling context. *Context-sensitive* analyses distinguish between different calling contexts. At the other end of the spectrum, *context-insensitive* analyses compute information only once and subsequently use the same information at all calling sites. Clearly, a context-sensitive analysis is more precise than a context-insensitive analysis, but it is also more costly (Nielson, Nielson, and Hankin, 1999). The choice between which technique to use amounts to a careful balance between precision and efficiency (Nielson, Nielson, and Hankin, 1999). The dependency analysis presented in the previous chapter is an interprocedural, flow-sensitive, context-insensitive data-flow analysis. Regarding *pure* context-sensitivity, in a functional language such as αSmil, in which predicate calls and the manipulation of the returned outputs are omnipresent, unfolding predicates at each call site and recomputing the needed information seems to be a daunting perspective that risks becoming prohibitive in terms of execution time very quickly. On the other hand, choosing to analyse predicates in isolation and to dispense completely with information regarding

the calling context leads to clear precision losses as illustrated in Section 5.4.1 and discussed in Section 5.4.2. In order to address this aspect, we have devised a solution based on *symbolic dependencies* that requires an extension of our abstract dependency domain (Definition 5.2.1) but which otherwise has a minimal impact on the dependency analysis at an intraprocedural and interprocedural level.

Outline. In this chapter we present our solution based on symbolic dependencies. We start by illustrating the addressed problem and the desired results in Section 6.2. In Section 6.3 and Section 6.4, we present the extended abstract dependency domain. We show the insertion and use of symbolic components at the intra- and interprocedural level of our dependency analysis in Section 6.5 and Section 6.6, respectively. Finally, we discuss their impact on the precision of the computed dependency information.

6.2 Symbolic Dependency Components in a Nutshell

Symbolic dependency components allow us to compute interprocedural predicate summaries with lazy components, in which the caller's intraprocedural information and context can be injected on an as-needed basis. The interprocedural dependency information for each predicate is still computed only *once* and propagated back to every possible call site. However, even though the analysis does not systematically recompute the dependency for the called predicate, it shows a form of *context-sensitivity* (Hind, 2001) and leads to increased precision, by creating *templates* with symbolic elements for each predicate. These elements introduce degrees of freedom in our interprocedural dependencies and allow us to parameterize and vary them according to the caller's actual intraprocedural context. Thus, we exclude some sources of coarse over-approximations without sacrificing scalability.

Previously, in Section 5.4.1 we illustrated on two αSmil example predicates, `thread` and `start_address`, how failing to take into consideration the current context of a caller leads to over-approximations. We argued in Section 5.4.2 that a more precise dependency blueprint can emerge, once we consider a predicate's use as well. The first example predicate given in Chapter 5, `thread`, is an *accessor* predicate: it receives a process `p` and an index `i` as inputs and returns the `i`-th associated thread of the process `p` when executing succesfully, i.e. when exiting with the **true** label. The computed predicate's dependency summary for the successful execution scenario was the following:

$$\begin{array}{lcl} \mathrm{p} & \mapsto & \{\ \mathrm{threads} \mapsto \langle \oslash \ \triangleright\ \mathrm{i}\text{: } [\mathrm{Some} \mapsto \top;\ \mathrm{None} \mapsto \bot] \rangle \} \\ \mathrm{i} & \mapsto & \top \end{array}$$

This dependency information is expressive: it shows that only one of the 4 fields of the input process is read by the predicate, while all others are irrelevant for its output. The read field, `threads`, corresponds to the array of threads associated to the input process `p`. Furthermore, the dependency summary shows that for this array only the `i`-th element is inspected. This element is entirely needed while all others are irrelevant.

This summary provides a rather detailed and precise blueprint of the predicate's output dependencies on its inputs. Yet, it fails to make one subtle, but important distinction regarding the dependency on the `i`-th element of the associated `threads` array. If we want to be more accurate while describing this predicate's dependency, we need to acknowledge that the predicate itself is not actually needing or depending on the `i`-th associated thread of the process. Indeed, it does not read or use it *per se*, it merely retrieves it. Thus, the dependency on the input process' `i`-th associated thread is *relative to* the amount in which the callers of the `thread` predicate will use the output element in which it is retrieved. It is important to distinguish between these two rather subtle nuances. Failing to do so can shadow information that is computed while analysing callers of the `thread` predicate. This was exactly what happened for our second example predicate, `start_address`. The predicate `start_address` receives a process `p` and an index `j` as inputs. It makes a call to the predicate `thread`, thus reading the `j`-th associated element of the process `p`. If this is an active element, it further accesses the field `stack`, from which it only reads the start address, `start`. The *obtained* dependency result:

$$\begin{array}{lcl} \mathrm{p} & \mapsto & \{\ \mathrm{threads} \mapsto \langle \oslash \ \triangleright\ \mathrm{j}\text{: } [\mathrm{Some} \mapsto \top;\ \mathrm{None} \mapsto \bot]\rangle\} \\ \mathrm{j} & \mapsto & \top \end{array}$$

was an over-approximation of the *desired* dependency result:

$$\begin{array}{lcl} \mathrm{p} & \mapsto & \{\ \mathrm{threads} \mapsto \langle \oslash \ \triangleright\ \mathrm{j}\text{: } [\mathrm{Some} \mapsto \{t \mapsto \{\mathrm{stack} \mapsto \{\mathrm{start} \mapsto \top\}\}\};\ \mathrm{None} \mapsto \bot]\rangle\} \\ \mathrm{j} & \mapsto & \top \end{array}.$$

Intraprocedurally, the dependency analysis was correctly detecting that only the field `stack` of the thread was needed, for which only the `start` field was read. However, when joining the dependency information computed locally for `start_thread` with the one given by the predicate's `thread` dependency summary, we obtain less precise dependency results. This scenario is not a corner case; it would typically be exhibited in the case of accessor predicates and their callers.

In order to address this source of precision loss, we can introduce symbolic or lazy components in our abstract dependency domain. As a first attempt and approximation, we could consider the set of output variables of a predicate as the lazy components. These can be seen as the points at which a caller predicate may insert its intraprocedural information in the dependency summary computed for the callee predicate.

The dependency summary for a successful execution of the `thread` predicate, i.e. the **`true`** exit label, would therefore not map the `i`-th element of the `threads` array to *everything*, i.e. $\top$, the top element of our abstract dependency domain. Instead, this would be mapped to the symbolic set of output variables in which this input subelement is retrieved, i.e. the set containing the `ti` output variable. We denote this set by **Deferred**(ti) as it represents the set of points in which a caller predicate can inject its context. Establishing the dependency on the `i`-th associated thread of the input process `p` is thus *deferred* or postponed and left to the caller predicates; it is relative to their context and the amount in which they use the output `ti`:

$$\begin{array}{lll} \mathrm{p} & \mapsto & \{\ \mathrm{threads} \mapsto \langle \oslash \ \triangleright\ \mathrm{j}\colon [\mathrm{Some} \mapsto \{\ \mathrm{t} \mapsto \textbf{Deferred}(\mathrm{ti})\}; \mathrm{None} \mapsto \bot]\rangle\} \\ \mathrm{j} & \mapsto & \top \end{array}.$$

Using this dependency summary when computing the information for the predicate `start_thread` we would obtain the targeted dependency result:

$$\begin{array}{lll} \mathrm{p} & \mapsto & \{\ \mathrm{threads} \mapsto \langle \oslash \ \triangleright\ \mathrm{j}\colon [\mathrm{Some} \mapsto \{t \mapsto \{\mathrm{stack} \mapsto \{\mathrm{start} \mapsto \textbf{Deferred}(\mathrm{adr})\}\}\}; \mathrm{None} \mapsto \bot]\rangle\} \\ \mathrm{j} & \mapsto & \top \end{array}.$$

This dependency summary for `start_address` shows that the dependency on the `j`-th associated thread of the input process `p`, depends on the amount in which the output `adr`, representing the start address of the thread's stack, is subsequently used. Indeed, `start_address` itself is an accessor predicate.

This first approximation of lazy components as sets of output variables of a predicate is effective for accessor predicates. However, its limitations become visible when considering functional, non-destructive *mutator* predicates, for example. Such predicates receive a compound input, destructure it, and construct a *new* output variable. This is created by modifying only one of the compound input's subelements, and by copying all the rest without further changes. For example, the predicate `set_thread`, shown below, is the dual of our `thread` example predicate. It receives a process `p`, a thread `ti` and an index `i` as inputs, and returns a new process `r` as an output, obtained by setting the `i`-th associated thread in the `threads` array to `ti`, and by copying everything else from `p`.

```
predicate set_thread (process p, int i, thread ti)
-> [true: process r]
{{array<option<thread>> threads, option<thread> tio}} {
  r := p : [true -> 1];
  threads := r.threads : [true -> 2];
  tio := Some(ti) : [true -> 3];
  threads := [threads with i = tio] : [true -> 4, false -> 6];
  r := {r with threads = threads} : [true -> 5];
  [true];
  [error]
}
```

The dependency summary computed for this predicate on the exit label **true** is shown below. It indicates that the given inputs, the index `i` and the thread `ti`, used for updating the `i`-th associated thread of the output process `r`, are completely needed. For the input process `p`, the fields `pid`, `crt_thread` and `adr_space` are completely needed as well. They are copied without further changes to the output `r`. From the array of associated threads, all elements, except the `i`-th, are needed as well. The latter is completely irrelevant since it is replaced in the output `r` by the given `ti`. The former are simply read and copied to `r`:

$$\begin{array}{lcl} \mathrm{p} & \mapsto & \left\{ \begin{array}{cc} \mathrm{threads} & \mapsto \langle \top \triangleright \mathrm{i}\colon \oslash \rangle \\ \mathrm{pid} & \mapsto \top \\ \mathrm{crt\_thread} & \mapsto \top \\ \mathrm{adr\_space} & \mapsto \top \end{array} \right\} \\ \mathrm{i} & \mapsto & \top \\ \mathrm{ti} & \mapsto & \top \end{array}$$

At a first glance, this dependency summary seems to reflect rather accurately, the predicate's inputs and input subelements on which the output process `r` depends on. However, similarly to the accessor predicate `thread`, a further distinction is possible. The predicate `set_thread` does not depend itself on the input `ti`, nor on the fields of the process `p`. It does not use these for new computations – it simply copies them to the corresponding output subelements. Just as before, the amount in which the output's subelements are used subsequently characterizes more precisely the dependency on the inputs of `set_thread`. For instance, the dependency on `p`'s current thread field should be the symbolic element corresponding to the output's process `crt_thread`. However, our first attempt at representing symbolic elements as sets of output variables seen as a whole, does not allow us to convey such information. For expressing it, we first need to be able to refer to the substructure `r.crt_thread` and use this as a lazy component in which callers may inject their own context. Similarly, for the `threads` array we need to be able to refer to all other elements except the `i`-th one. Thus, at the symbolic dependencies level as well, we need the capability of distinguishing between the different subelements of the inputs. This would allow us to obtain the following dependency summary:

$$\begin{array}{lcl} \mathrm{p} & \mapsto & \left\{ \begin{array}{cc} \mathrm{threads} & \mapsto \langle\ \mathbf{Deferred}(\mathrm{r.threads}\langle * \setminus \mathrm{i}\ \rangle) \triangleright \mathrm{i}\colon \oslash \rangle \\ \mathrm{pid} & \mapsto \mathbf{Deferred}(\mathrm{r.pid}) \\ \mathrm{crt\_thread} & \mapsto \mathbf{Deferred}(\mathrm{r.crt\_thread}) \\ \mathrm{adr\_space} & \mapsto \mathbf{Deferred}(\mathrm{r.adr\_space}) \end{array} \right\} \\ \mathrm{i} & \mapsto & \top \\ \mathrm{ti} & \mapsto & \mathbf{Deferred}(\mathrm{r.threads}\langle\ \mathrm{i}\ \rangle @ \mathrm{Some.t}) \end{array}$$

One way to capture the actual effect that is due to `set_thread` consists in replacing all deferred dependencies with $\oslash$, i.e. *nothing*, and simplifying the summary. The dependency summary thus obtained shows the dependency on `set_thread`'s inputs in the extreme case of calling the predicate and throwing away its result. In this case, the summary for `set_thread` would show that the predicate only depends on the input `i` and on the length or support of the `threads` array, captured by $\langle\oslash\rangle$. On the contrary, by replacing the deferred dependencies with $\top$, i.e. *everything*, we obtain exactly the results computed by the context-insensitive dependency analysis presented in Chapter 5. The information thus obtained shows the dependency on `set_thread`'s inputs when considering the other end of the spectrum, namely calling the predicate and using its result entirely.

The dependency summary with deferred occurrences is indeed precise. Not only does it create a dependency template in which callers can inject their own context, but it also distills the predicate's `set_thread` specification. A quick glance and interpretation of it indicates that it is indeed a non-destructive mutator, updating the `i`-th associated thread of a process to `ti` and preserving everything else.

In order to obtain such dependency summaries, we need to refine our first approximation of symbolic elements as sets of a predicate's output variables. Just as needed in our initial abstract dependency domain, we must reflect the layered structure of algebraic data types and arrays at the level of symbolic dependencies as well. To this end, we need to consider not only sets of output variables, but also symbolic *paths* to substructures within them.

6.3 Symbolic Paths

6.3.1 Symbolic Path Type

In order to extend our abstract dependency domain with symbolic dependencies and to obtain expressive dependency summaries as the ones discussed in the previous section, we begin by introducing *symbolic paths*. These are meant to mirror the layered structure of algebraic data types and arrays at the level of symbolic dependencies.

Each *deferred occurence* in a dependency summary is identified by *symbolic paths*. Symbolic paths are rooted at one of the program's variables and represent sequences of *symbolic* internal accesses inside some value's structure, i.e. they are symbolic traversals from one value to some of its subparts. Paths are chains of symbolic accesses leading to nested elements in which different calling contexts can be subsequently injected. We define a recursive type π of symbolic paths encompassing this.

Definition 6.3.1. Symbolic path type $\pi \in \Pi$.

$$
\begin{array}{lll}
\pi \in \Pi, \quad \pi := & \mid \varepsilon & \text{endpoint – root} \\
& \mid .f\pi & f \in \mathcal{F}, \\
& \mid @C\pi & C \in \mathcal{C}, \\
& \mid \langle i \rangle \pi & i \text{ index}, \\
& \mid \langle * \setminus i \rangle \pi & i \text{ index}, \\
& \mid \langle * \rangle \pi &
\end{array}
$$

An *endpoint*, denoted by ε, is the special path denoting an entire element. For structures, we denote the symbolic path to some field f by $.f\pi$. Similarly, for variants, we denote the path to some chosen constructor C by $@C\pi$. For arrays, we distinguish between three cases:

- symbolic paths referring to a specific array cell, identified by the cell's index i, and denoted by $\langle i \rangle \pi$;
- symbolic paths referring to all but one specific array cell, identified by its index i, and denoted by $\langle * \setminus i \rangle \pi$;

- symbolic paths referring to all the cells of an array, denoted by $\langle * \rangle \pi$.

With one exception, these symbolic paths directly reflect the cases of our abstract dependency domain. For instance, the correspondance between symbolic paths for structures or variants is immediately apparent. In contrast, for arrays, the abstract dependency domain included two cases, namely $\langle \delta \rangle$, corresponding to a dependency applying to all of the cells, and $\langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle$, corresponding to arrays with a general dependency applying to all but one exceptional cell, for which a specific dependency is known. In order to reflect the second case in the deferred occurrences, we need to be able to refer to the exceptional cell on one hand, and to all other cells of the array on the other hand. Hence, to this end, we need to introduce two symbolic path types: the symbolic $\langle i \rangle \pi$ path for expressing deferred occurrences of exceptional cells, and the $\langle * \setminus i \rangle \pi$ symbolic path for expressing deferred occurences of all the other array cells, except the one identified by i.

The action of appending a non-empty path π' to another path π is denoted by $\pi :: \pi'$. We call :: the *extension operator* and when applying it we say that we *extend* π with π'.

We further consider *sets* $P \subset \Pi$ of symbolic paths π and define the partial order $\overset{\circ}{\sqsubseteq}$ between them.

Definition 6.3.2. Partial Order $\overset{\circ}{\sqsubseteq}$ for Path Sets.

$$\forall P \subset \Pi, P' \subset \Pi, \ P \overset{\circ}{\sqsubseteq} P' \iff P \subseteq P'.$$

They establish a semi-lattice based on the *subset order*. The bottom element of this semi-lattice is $\emptyset$, the empty set of paths:

$$\forall P \subset \Pi, \ \emptyset \overset{\circ}{\sqsubseteq} P.$$

There is no *top* element. Theoretically, this would correspond to the set representing *all* possible paths. In practice, this cannot be constructed and we chose not to add a special case for it to our symbolic path type π.

The *join* operation of deferred path sets is based on *set union* and is denoted by $\overset{\circ}{\vee}$.

Definition 6.3.3. Join Operation $\overset{\circ}{\vee}$ for Path Sets.

$$\forall P \subset \Pi, P' \subset \Pi, \ P \overset{\circ}{\vee} P' = P \cup P'.$$

It is *symmetric* and the value obtained by joining two path sets is the *least upper bound*.

Applying the extension operator :: on a set of symbolic paths P amounts to a pointwise extension of each member of the path set.

Definition 6.3.4. Extension Operator $\overset{\circ}{::}$ for Path Sets.

$$\forall P \subset \Pi, \ P \overset{\circ}{::} \pi' = \{\pi :: \pi' | \ \pi \in P\}.$$

6.3.2 Semantics of Symbolic Paths

Semantically, paths of type π defined previously, are a symbolic representation of several *actual* paths. In the following, we explicit this notion and we begin by defining simple, *actual* paths in a value of the universe $\mathbb{D}$ (Definition 4.4.1).

Actual paths represent a unique sequence of internal accesses inside some value's structure, leading to a *single* nested element. Unlike symbolic paths that can, for instance, cover multiple elements of an array, an actual path designates a single subvalue of a structure, variant or array. The recursive actual path type $\widetilde{\pi} \in \widetilde{\Pi}$ is defined below.

Definition 6.3.5. Actual Path Type $\widetilde{\pi} \in \widetilde{\Pi}$.

$$\begin{array}{rll} \widetilde{\pi} := & \mid \widetilde{\varepsilon} & \text{empty,} \\ & \mid \widetilde{.f}\widetilde{\pi} & f \in \mathcal{F}, \\ & \mid \widetilde{@C}\widetilde{\pi} & C \in \mathcal{C}, \\ & \mid \widetilde{\langle i \rangle}\widetilde{\pi} & i \text{ index.} \end{array}$$

A symbolic path π *covers* an actual path $\widetilde{\pi}$ if, when given a *valuation* E (Definition 4.4.2) of the index variables for arrays, it matches $\widetilde{\pi}$. A set of symbolic paths covers an actual path $\widetilde{\pi}$ if at least one of the symbolic paths matches $\widetilde{\pi}$. We denote this by the $\asymp_E$ relation that is parameterized by a valuation E. The definition of $\asymp_E$ is given in Table 6.1.

TABLE 6.1 – $\asymp_E$ – Path Semantics

$$\frac{}{\varepsilon \asymp_E \widetilde{\varepsilon}} \asymp_E \widetilde{\varepsilon}$$

$$\frac{\pi \asymp_E \widetilde{\pi}}{.f\pi \asymp_E \widetilde{.f}\widetilde{\pi}} \asymp_E \textsc{Struct} \qquad \frac{\pi \asymp_E \widetilde{\pi}}{@C\pi \asymp_E \widetilde{@C}\widetilde{\pi}} \asymp_E \textsc{Var}$$

$$\frac{\pi \asymp_E \widetilde{\pi} \qquad E(i) = j}{\langle i \rangle \pi \asymp_E \widetilde{\langle j \rangle}\widetilde{\pi}} \asymp_E \textsc{Cell}$$

$$\frac{\pi \asymp_E \widetilde{\pi}}{\langle * \rangle \pi \asymp_E \widetilde{\langle j \rangle}\widetilde{\pi}} \asymp_E \textsc{AnyCell} \qquad \frac{\pi \asymp_E \widetilde{\pi} \qquad E(i) \neq j}{\langle * \setminus i \rangle \pi \asymp_E \widetilde{\langle j \rangle}\widetilde{\pi}} \asymp_E \textsc{OutCell}$$

Given a valuation E, a set P of symbolic paths covers an actual path $\widetilde{\pi}$ if at least one of the symbolic paths in the set covers or matches $\widetilde{\pi}$:

$$\forall P \subset \Pi,\ P \overset{\circ}{\asymp}_E \widetilde{\pi} \iff \exists \pi \in P,\ \pi \asymp_E \widetilde{\pi}.$$

The interpretation $[\![P]\!]^E$ of a set of paths P is then the set of single actual paths that are covered, given a valuation E.

Definition 6.3.6. Interpretation $[\![P]\!]^E$ of a set of paths P.

$$\forall P \subseteq \Pi,\ [\![P]\!]^E = \{\widetilde{\pi} |\ P \overset{\circ}{\asymp}_E \widetilde{\pi}\}.$$

The partial order $\overset{\circ}{\sqsubseteq}$ (Definition 6.3.2) on sets of paths is compatible with the interpretation $[\![P]\!]^E$ in the sense that when $P \overset{\circ}{\sqsubseteq} Q$ holds, the interpretation $[\![P]\!]^E$ of P is included in $[\![Q]\!]^E$ for every valuation.

$$\forall P, Q \subseteq \Pi, \forall E, P \overset{\circ}{\sqsubseteq} Q \iff [\![P]\!]^E \subseteq [\![Q]\!]^E.$$

Each single path can be interpreted as a way to find a subpart of a value, which we explicit by the following function at. It is not defined for all cases, since not all paths can be applied to all values.

Definition 6.3.7. Function at.

$$\mathsf{at} : \widetilde{\Pi} \times \mathbb{D} \to \mathbb{D}$$

$$\mathsf{at}(\widetilde{\pi}, v) \quad = \quad \begin{cases} v & \text{when } \widetilde{\pi} = \widetilde{\varepsilon} \\ \mathsf{at}(\widetilde{\pi}', v_i) & \text{when } \widetilde{\pi} = \widetilde{.f_i}\widetilde{\pi}' \text{ and} \\ & v = \{f_1 = v_1, \ldots, f_i = v_i, \ldots, f_n = v_n\} \\ \mathsf{at}(\widetilde{\pi}', v_C) & \text{when } \widetilde{\pi} = \widetilde{@C_i}\widetilde{\pi}' \text{ and} \\ & v = C_i[v_C] \\ \mathsf{at}(\widetilde{\pi}', v_i) & \text{when } \widetilde{\pi} = \widetilde{\langle i \rangle}\widetilde{\pi}' \text{ and} \\ & v = (\mathcal{P}, (v_k)_{k \in \mathcal{P}}), \\ & i \in \mathcal{P} \end{cases}$$

6.3.3 Well-Typed Paths and Path Sets

Symbolic paths cannot be used in every context: their interpretation must be made in the context of a type τ. An *endpoint*, i.e. the ε symbolic path can apply to any type. In contrast, other symbolic paths that exhibit specific data features can only apply to the corresponding types. For instance, a path such as $.f\pi$ is meaningless on values which are not records, or on record values that do not exhibit a field f, the field specified in the symbolic path.

A path set can be seen as a set of sequences of internal accesses inside some values's structure. In that sense, it is a set of possible traversals from one value to some of its subparts. To characterize the contexts in which a path set is *well-typed*, we need to consider the types of values to which it can be applied and the types of values to which it can lead to. Therefore, in the following, we begin by defining a typing judgement for symbolic paths as a three-place relation $\pi : \tau \to \tau'$, whose meaning is that π can be applied to any value of type τ and in that case it will always describe subvalues of type

τ'. Additionally, the typing judgement is also parameterized by a set of *input variables* $\mathcal{I}$, which are the variables having the right to appear as identifiers for array accesses. This is detailed in Table 6.2.

$$\frac{}{\mathcal{I} \vdash \varepsilon \ : \ \tau \ \rightarrow \ \tau} \ \text{WT}\varepsilon$$

$$\frac{\tau = \textbf{\textit{struct}}\{f_1 : \tau_1, \ldots, f_i : \tau_i, \ldots, f_n : \tau_n\} \quad \mathcal{I} \vdash \pi_i : \tau_i \rightarrow \tau'}{\mathcal{I} \vdash .f_i\pi_i \ : \ \tau \ \rightarrow \ \tau'} \ \text{WTStructPath}$$

$$\frac{\tau = \textbf{\textit{variant}}[C_1 : \tau_1 | \ \ldots | \ C_i : \tau_i | \ \ldots | \ C_n : \tau_n] \quad \mathcal{I} \vdash \pi_C : \tau_i \rightarrow \tau'}{\mathcal{I} \vdash @C_i\pi_C \ : \ \tau \ \rightarrow \ \tau'} \ \text{WTVarPath}$$

$$\frac{\Gamma \vdash \pi \ : \ \tau \ \rightarrow \ \tau'}{\mathcal{I} \vdash \langle * \rangle \pi \ : \ \textbf{\textit{arr}}^{\tau_i}\langle \tau \rangle \ \rightarrow \ \tau'} \ \text{WTArrayPath}$$

$$\frac{\mathcal{I} \vdash \pi \ : \ \tau \ \rightarrow \ \tau' \qquad \mathcal{I}(i) = \tau_i}{\mathcal{I} \vdash \langle i \rangle \pi \ : \ \textbf{\textit{arr}}^{\tau_i}\langle \tau \rangle \ \rightarrow \ \tau'} \ \text{WTCellPath}$$

$$\frac{\mathcal{I} \vdash \pi \ : \ \tau \ \rightarrow \ \tau' \qquad \mathcal{I}(i) = \tau_i}{\mathcal{I} \vdash \langle * \setminus i \rangle \pi \ : \ \textbf{\textit{arr}}^{\tau_i}\langle \tau \rangle \ \rightarrow \ \tau'} \ \text{WTOutPath}$$

Table 6.2 – Well-Typed Dependency Paths

A set P of symbolic paths is *well-typed* if every path contained by it is well-typed for the same types:

$$\forall P \subset \Pi, \ \mathcal{I} \overset{\circ}{\vdash} P : \tau \rightarrow \tau' \iff \forall \pi \in P, \ \mathcal{I} \vdash \pi : \tau \rightarrow \tau'.$$

The *well-typedness* property of sets of symbolic paths is preserved by the join operation $\overset{\circ}{\vee}$ (Definition 6.3.3):

$$\forall P', P'' \in \Pi, \ \forall \tau', \tau'' \in \mathbb{T},$$
$$\mathcal{I} \overset{\circ}{\vdash} P' : \tau' \rightarrow \tau'' \Rightarrow \mathcal{I} \overset{\circ}{\vdash} P'' : \tau' \rightarrow \tau'' \Rightarrow \mathcal{I} \overset{\circ}{\vdash} P' \overset{\circ}{\vee} p'' : \tau' \rightarrow \tau''.$$

When extending a well-typed set of symbolic paths with a well-typed path using the extension operator $\overset{\circ}{::}$ (Definition 6.3.4), the resulting set of symbolic paths is well-typed

as well:

$$\forall P' \in \Pi,\ \forall \tau, \tau', \tau'' \in \mathbb{T},$$
$$\mathcal{I} \overset{\circ}{\vdash} P' : \tau' \to \tau'',\ \ \mathcal{I} \vdash \pi' : \tau'' \to \tau \quad \Rightarrow \quad \mathcal{I} \overset{\circ}{\vdash} P' :: \pi' : \tau' \ \to\ \tau.$$

6.4 Abstract Dependency Domain with Deferred Accesses

Frequently, as explained in Section 6.2, the dependency on a predicate's input variable is relative to the amount in which some of the predicate's outputs are subsequently needed. More precisely, these outputs are those into which the input variable is copied and retrieved. We strive to avoid over-approximations in such cases and to create degrees of freedom for the callers by treating such output variables as points in which callers can inject their own context externally. In other words, we want to *defer* the computation of the dependency on certain input variables of a predicate to the predicate's callers, since they have additional information about the actual use of the predicate's outputs.

In our previous section — Section 6.3 — we have introduced and defined an intermediate level consisting of symbolic paths and path sets. These reflect the layered structure of algebraic data types and arrays, and allow us to consider not only output variables as a whole, but also symbolic paths within them. Thus, we can compute more flexible and expressive dependency summaries, with finer-grained elements. We can finally link these two ideas and extend our abstract dependency domain with deferred dependencies, by including an additional dependency case in our domain $\delta \in \mathcal{D}$, initially defined (Definition 5.2.1) in Section 5.2.

Definition 6.4.1. Extended Abstract Dependency Domain $\delta \in \mathcal{D}$.

$\delta :=$	$\mid \top$	*Everything* – atomic case	(i)
	$\mid \oslash$	*Nothing* – atomic case	(ii)
	$\mid \bot$	*Impossible* – atomic case	(iii)
	$\mid \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}$	$f_1, \ldots, f_n$ fields	(iv)
	$\mid [C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m]$	$C_1, \ldots, C_m$ constructors	(v)
	$\mid \langle \delta \rangle$		(vi)
	$\mid \langle \delta_{def} \triangleright i \ :\ \delta_{exc} \rangle$	i array index	(vii)
	$\mid$ **Deferred**$(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\})$	deferred accesses	(viii)

A deferred dependency, shown in (viii) consists of a mapping which binds output variables, which we also call *root variables* in this case, to sets of symbolic paths.

Definition 6.4.2. Access Map.

$$\mathcal{A} : \mathcal{V} \twoheadrightarrow \Pi.$$

Only output variables can be treated as lazy dependency components. The sets of symbolic paths mapped to them, allow us to distinguish between their subelements. In the following discussion we will denote an access map $\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\}$ by a.

For the partial order $\sqsubseteq$ (Definition 5.2.2), defined in Chapter 5 and detailed in Table 5.1, an additional rule (Def) for comparing instances of deferred dependencies is added. This is shown in Table 6.3. The top and bottom elements of our dependency domain are as before $\top$ and $\bot$, respectively. Thus, any instance of a deferred dependency is more precise than $\top$ and less precise than $\bot$. Just as $\top$, $\bot$ and the special dependency case $\oslash$, a deferred dependency can be used in association to any type, albeit with some constraints for its elements.

$$\frac{\forall o \mapsto P \ \in \ a, \qquad a(o) \overset{\circ}{\sqsubseteq} a'(o)}{\textbf{Deferred}(a) \sqsubseteq \textbf{Deferred}(a')} \ \textsc{Def}$$

Table 6.3 – Extended Leq - Comparison of Two Domains

However, unlike the atomic cases $\top$, $\bot$ and $\oslash$, deferred dependencies are not related to $\oslash$ or to dependencies corresponding to structures, variants or arrays. Since they act as placeholders for dependencies that are effectively computed subsequently, instances of deferred dependencies can be compared only to $\top$ and $\bot$ or to other instances of deferred dependencies. For instance, comparing a deferred dependency to $\oslash$ would yield:

$$\textbf{Deferred}(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\}) \not\sqsubseteq \oslash$$
and
$$\oslash \not\sqsubseteq \textbf{Deferred}(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\}).$$

The extended join operation $\vee$ (Definition 5.2.3), initially defined in Section 5.2.1 and detailed in Table 5.2, is shown below in Table 6.4. It still has $\bot$ as its *identity* element and $\top$ as its absorbing element. Joining two instances of deferred dependencies amounts to a pointwise join of the path sets mapped to each output variable in the access maps. The join between an instance of a deferred dependency and a dependency corresponding to a structure, a variant, an array or to the special case $\oslash$, amounts to $\top$, the top element of our domain. Since we cannot make any supposition regarding deferred dependencies, we are forced to make a pessimistic assumption and to approximate to the least precise value. Join is a commutative operation for which the undisplayed cases in Table 6.4 are defined with respect to their symmetrical counterparts.

Similarly to join, the reduction operation $\oplus$ (Definition 5.2.4) has been initially defined in Section 5.2.1 and it has been detailed in Table 5.3. The extended form is shown in Table 6.5. It still has $\oslash$ as an *identity* element and $\bot$ as an *absorbing* element. When applying the reduction operation between a deferred dependency and a dependency δ' corresponding to a structure, a variant or an array, we over-approximate the deferred dependency to $\top$ and apply the reduction operation between δ' and $\top$. Applying the reduction operation between a deferred dependency and $\top$ behaves similarly; the outcome in this case is straightforward and amounts to $\top$. As was the case for join, applying the reduction operation between two instances of deferred dependencies

δ'		δ''		$\delta' \vee \delta''$
Deferred(a)	$\vee$	**Deferred**(a')	$=$	**Deferred**(a'') where $a''(o) = \begin{cases} a(o) \mathbin{\mathring{\vee}} a'(o) & \text{when } o \mapsto P_o \in a, o \mapsto P'_o \in a' \\ P_o & \text{when } o \mapsto P_o \in a \\ P'_o & \text{when } o \mapsto P'_o \in a' \end{cases}$
Deferred(a)	$\vee$	$\{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}$	$=$	$\top$
Deferred(a)	$\vee$	$[C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m]$	$=$	$\top$
Deferred(a)	$\vee$	$\langle \delta \rangle$	$=$	$\top$
Deferred(a)	$\vee$	$\langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle$	$=$	$\top$
Deferred(a)	$\vee$	$\oslash$	$=$	$\top$

TABLE 6.4 – $\vee$ – Extended Join

amounts to a pointwise join of the path sets mapped to each output variable in the access maps. The reduction operation is commutative and the undisplayed cases in Table 6.5 are defined with respect to their symmetrical counterparts.

δ'		δ''		$\delta' \oplus \delta''$
Deferred(a)	$\oplus$	**Deferred**(a)	$=$	**Deferred**(a'') where $a''(o) = \begin{cases} a(o) \mathbin{\mathring{\vee}} a'(o) & \text{when } o \mapsto P_o \in a, o \mapsto P'_o \in a' \\ P_o & \text{when } o \mapsto P_o \in a \\ P'_o & \text{when } o \mapsto P'_o \in a' \end{cases}$
Deferred(a)	$\oplus$	$\top$	$=$	$\top$
Deferred(a)	$\oplus$	$\{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}$	$=$	$\top \oplus \{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\}$
Deferred(a)	$\oplus$	$[C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m]$	$=$	$\top \oplus [C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m]$
Deferred(a)	$\oplus$	$\langle \delta \rangle$	$=$	$\top \oplus \langle \delta \rangle$
Deferred(a)	$\oplus$	$\langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle$	$=$	$\top \oplus \langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle$

TABLE 6.5 – $\oplus$ – Extended Reduction Operator

Finally, the extractions previously defined for dependencies δ (Definition 5.2.5, 5.2.6, 5.2.7, 5.2.8 and 5.2.9) have been extended in order to handle deferred dependencies as well. Their treatment is summarized in Table 6.6. Making array-specific extractions, as well as extracting field and constructor dependencies on a deferred dependency, amounts to a pointwise *extension* of every path set in the access map with the corresponding symbolic path.

Finally, we add the following rule to the well-typed dependency rules given in Chapter 5, Table 5.5:

Extraction	δ	Result
Field	$\textbf{Deferred}(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\}).f$	$\textbf{Deferred}(o_1 \mapsto P_1 \overset{\circ}{::} .f\varepsilon; \ldots; o_k \mapsto P_k \overset{\circ}{::} .f\varepsilon;)$
Constructor	$\textbf{Deferred}(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\})@C$	$\textbf{Deferred}(o_1 \mapsto P_1 \overset{\circ}{::} @C\varepsilon; \ldots; o_k \mapsto P_k \overset{\circ}{::} @C\varepsilon;)$
Cell	$\textbf{Deferred}(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\})\langle i\rangle$	$\textbf{Deferred}(o_1 \mapsto P_1 \overset{\circ}{::} \langle i\rangle\varepsilon; \ldots; o_k \mapsto P_k \overset{\circ}{::} \langle i\rangle\varepsilon;)$
Array General	$\textbf{Deferred}(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\})\langle * \rangle$	$\textbf{Deferred}(o_1 \mapsto P_1 \overset{\circ}{::} \langle * \rangle\varepsilon; \ldots; o_k \mapsto P_k \overset{\circ}{::} \langle * \rangle\varepsilon;)$
Outside Cell	$\textbf{Deferred}(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\})\langle * \setminus i\rangle$	$\textbf{Deferred}(o_1 \mapsto P_1 \overset{\circ}{::} \langle * \setminus i\rangle\varepsilon; \ldots; o_k \mapsto P_k \overset{\circ}{::} \langle * \setminus i\rangle\varepsilon;)$

TABLE 6.6 – Extended Extraction Operators

$$\frac{\begin{array}{cc} \Gamma(o_1) = \tau_1 & \Gamma, \mathcal{I} \overset{\circ}{\vdash} P_1 \ : \ \tau_1 \ \to \ \tau \\ & \ldots \\ \Gamma(o_k) = \tau_k & \Gamma, \mathcal{I} \overset{\circ}{\vdash} P_k \ : \ \tau_k \ \to \ \tau \\ o_1 \in \mathcal{O} & \ldots \qquad o_k \in \mathcal{O} \end{array}}{\Gamma, \mathcal{I}, \mathcal{O} \vdash \textbf{Deferred}(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\}) \ : \ \tau} \ \text{WTDEFERRED}$$

TABLE 6.7 – Well-Typed Dependencies – Extended

6.5 Deferred Dependencies at the Intraprocedural Level

6.5.1 Extended Intraprocedural Dependency Analysis

At the intraprocedural and interprocedural level of our dependency analysis, the introduction of deferred dependencies has a minimal impact in terms of required changes.

Intraprocedurally, each predicate is analysed on every possible exit label. As explained in Section 5.3.2, our dependency analysis is a *backward* data-flow analysis. For each possible exit label of a predicate, the control flow graph is traversed backwards, starting from the exit node that corresponds to the analysed execution scenario. Dependency information is computed at every point of the control flow graph, for each of the predicate's input, output and local variables and this information is gradually refined until a fixed point is reached.

By traversing the control flow graph backwards, we take advantage of the information regarding the outputs that are associated to the analysed exit label and we consider only the relevant ones starting from the initialisation phase. As explained previously in Section 5.3.2, the intraprocedural domain for the currently analysed exit label is initialised with its associated output variables mapped to $\top$, the least precise element of our abstract dependency domain. This is a conservative over-approximation: it is considered that control on the outputs is lost and that these are entirely observed externally. As illustrated in Section 6.2, this over-approximation propagates along the control flow graph and, in certain cases, has a non-negligible impact on the precision of the computed dependency summaries.

We argued that at the intraprocedural level of the analysis, a subtle, but important distinction can be made, regarding the dependency on certain inputs. This consists in

distinguishing between the cases in which a predicate effectively uses an input subelement to compute an output subelement, and those in which it simply forwards it to an output subelement. In the latter cases, the predicate does not use or need such an input subelement per se, and as a consequence, the dependency on it is relative to the amount in which the predicate's callers will subsequently use the output in which it is retrieved. At the intraprocedural level, in order to avoid the propagation of over-approximations, it is important to make this distinction early on, from the initialisation phase. Therefore, we introduce deferred dependencies at this level, instead of mapping the output variables to $\top$ as was previously done.

For a predicate p of the following form:

$$p(e_1, \ldots, e_n)\ [\lambda_1 : o_{1,1}, \ldots, o_{1,k_1} \mid \ldots \mid \lambda_i : o_{i,1}, \ldots, o_{i,k_i} \mid \ldots \mid \lambda_m : o_{m,1}, \ldots, o_{m,k_m}]$$

analysed on the λ_i exit label, the intraprocedural dependency domain used for initialising the node corresponding to λ_i is the following:

$$\begin{array}{lll} o_{i,1} & \mapsto & \textbf{Deferred}(o_{i,1} \mapsto \{\varepsilon\}) \\ \ldots & \ldots & \ldots \\ o_{i,k_i} & \mapsto & \textbf{Deferred}(o_{i,k_i} \mapsto \{\varepsilon\}) \end{array}$$

For each associated output $o_{i,j},\ 1 \leq j \leq k_i$ of the analysed label λ_i, a set $P_{o_{i,j}}$ of symbolic paths is constructed. Initially, this consists of a single element, namely the ε path. The deferred dependency associated to each output $o_{i,j}$ is an access map binding $o_{i,j}$ itself to its corresponding set of symbolic paths $P_{o_{i,j}}$. Since the symbolic paths ε refer to the output variables in their entirety, this is still a conservative approximation, but, in contrast to our previous initialisation strategy, it acknowledges the fact that dependencies on the inputs might be relative to the amount in which the outputs are subsequently used. It allows injecting context-sensitive information later on.

This new initialisation strategy is enough to incorporate the expressive power of deferred dependencies at an intraprocedural level. Whereas before we were computing label-specific dependency summaries as input-output relations, the new strategy allows us to obtain label-specific dependency *templates* with lazy components, that can be parameterized and varied according to a caller's own intraprocedural context. These can be seen as context-insensitive dependency summaries with context-sensitive leaves.

6.5.2 Intraprocedural Dependency Analysis Illustrated

In order to illustrate the use of deferred dependencies at an intraprocedural level, we revisit our `thread` example predicate, discussed in Section 5.3.3. As done previously, we consider the **true** execution scenario and apply our extended dependency analysis. We initialize the dependency corresponding to the **true** exit node, by mapping the predicate's output `ti` to the deferred dependency mapping it to a set containing a single symbolic path, namely ε.

After the initialisation phase, the analysis continues as before, by traversing the control flow graph backwards and by applying at each step the corresponding data-flow equation. The deferred dependency is propagated upwards until the entry node is reached and analysed.

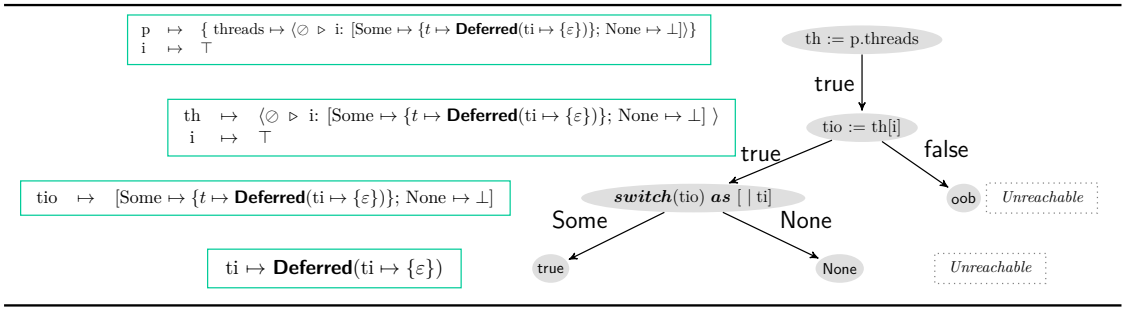


FIGURE 6.1 – Analysing `thread` – Dependency Summary with Deferred Occurrences

The final dependency summary for the **true** exit label of the predicate is obtained:

p ↦ { threads ↦ ⟨∅ ▷ i: [Some ↦ {t ↦ **Deferred**(ti)}; None ↦ ⊥]⟩}
i ↦ ⊤

and this is similar to the targeted dependency information for `thread`, discussed in Section 6.2 and illustrated on page 117.

6.6 Deferred Dependencies at the Interprocedural Level

At the interprocedural level, the impact of introducing deferred dependencies is visible only at the level of the substitutions that have to be performed. Previously, the only required substitution consisted in replacing all occurrences of formal input parameters of a predicate with the corresponding effective input parameters. After having introduced deferred dependencies, further substitutions are needed. These can be easily illustrated by revisiting our `start_address` example predicate discussed in Section 5.4.1. As done previously, we consider the **true** execution scenario and apply our extended dependency analysis.

We begin by initialising the output `adr` with a corresponding deferred dependency, as discussed in Section 6.5.1. The analysis traverses the control flow graph backwards and computes the dependency information at each node, until reaching the control flow graph's entry node, which corresponds to a call to the `thread` predicate. The intermediate dependency results are shown in Figure 6.2.

We obtain the dependency summary for the **true** exit label of the called predicate `thread`. In order to be able to use it, we must first substitute the formal input parameters, i.e. `p` and `i`, appearing in it, with the effective arguments of the call, i.e. `p` and `j`. Additionally, in deferred dependencies, we also have to substitute the formal output

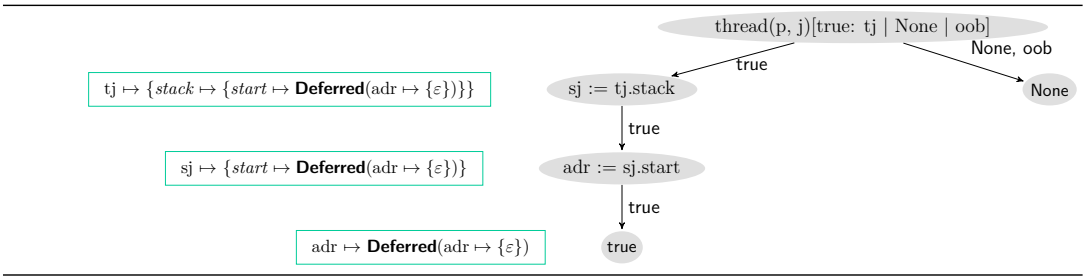


FIGURE 6.2 – $G_{start\_address}$ – Intermediate Dependency Results for `start_address`

parameters appearing as roots in the access maps, i.e. `ti`, with the corresponding effective output parameters. These substitutions are shown in Figure 6.3. Formal index variables appearing in dependencies corresponding to arrays have to be substituted with their effective counterparts as well. Similarly, any formal index variable appearing in symbolic paths that correpond to arrays must be substituted by the corresponding effective index variable.

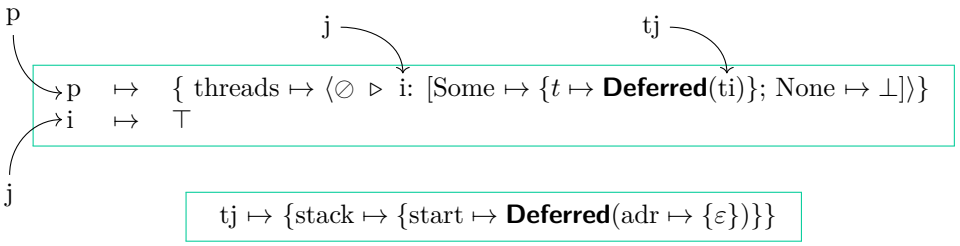
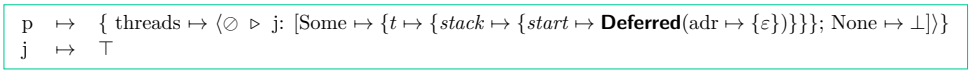


FIGURE 6.3 – Substitution of Formal Parameters by Effective Parameters

We can finally take advantage of the flexibility obtained using deferred dependencies by injecting the caller's intraprocedural dependency information into the deferred occurrences of the callee's dependency summary. This is another type of substitution and consists in replacing deferred occurrences of formal output parameters of a predicate by the dependency information computed in the current context for the corresponding effective output parameters. For our `start_address` example, this is shown in Figure 6.4 and amounts to substituting the dependency computed for `tj` in the deferred occurrence of `ti` in the dependency summary of `thread`.

After this substitution, we obtain the following dependency summary for the exit label **true** of the `start_address` predicate:



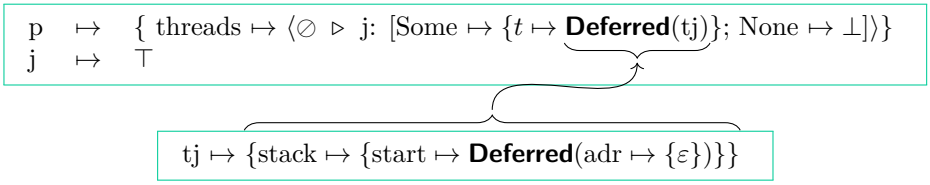


FIGURE 6.4 – Substituting Deferred Dependencies by Actual Dependencies

6.6.1 Applying Context-Sensitive Information by Substitution

As shown in our previous example, deferred dependencies associate sets of symbolic paths to certain root variables. We can substitute such deferred dependencies by actual dependencies computed in the current context, by applying the symbolic paths to the actual dependency to substitute. We iterate through entire dependency summaries in order to substitute the nested deferred dependencies appearing at some leaves. This substitution can be seen as an application of contextual information to summaries with deferred dependencies, which are essentially context-insensitive abstractions with context-sensitive leaves. It is denoted by a mapping σ which associates dependencies to root variables appearing in deferred access maps.

Definition 6.6.1. Substitution σ.

$$\sigma : \mathcal{V} \to \mathcal{D}.$$

Simultaneously, while substituting root variables in deferred dependencies by their actual dependencies, computed in the current intraprocedural context, we also substitute indices in information corresponding to arrays. These are substituted either by another array index, i.e. the one corresponding to an actual input parameter, or they are eliminated, when corresponding to a local variable. Their elimination consists in approximating the dependencies so as to remove references to the array index. This substitution is denoted by ϕ and it is a mapping from variables to new variables to replace them.

Definition 6.6.2. Substitution ϕ.

$$\phi : \mathcal{V} \nrightarrow \mathcal{V}.$$

The two substitutions can be done separately. However, for performance reasons, we chose to do them simultaneously. This is also what the actual implementation of the dependency analysis does. We denote the two simultaneous substitutions by $\blacktriangleleft\ (\sigma, \phi)$ and detail them in Table 6.9. Performing the two operations simultaneously can be seen as a manner of reinterpreting a dependency computed in one context in another context.

For sets of symbolic paths (as defined in Section 6.3.1) in deferred dependencies, the operation $P \bullet (\sigma(o), \phi)$ is the application of symbolic paths to the dependency of

the root variable o computed in the current context. For a deferred access map, all dependencies obtained by applying the symbolic paths are joined. The application of a symbolic path π to a dependency δ is denoted by $\pi \circ (\delta, \phi)$ and it is shown in Table 6.8. During the application, free variables appearing in symbolic paths associated to arrays are substituted by their corresponding index variables as given by ϕ. If ϕ does not contain a mapping for a free variable, an approximation is made in order to remove it and the dependency obtained by applying $\langle * \rangle$ is returned.

$\pi \circ (\delta, \phi)$		
$\varepsilon \circ (\delta, \phi)$	$=$	δ
$.f\pi \circ (\delta, \phi)$	$=$	$\pi \circ (\delta.f, \phi)$
$@C\pi \circ (\delta, \phi)$	$=$	$\pi \circ (\delta @C, \phi)$
$\langle * \rangle \pi \circ (\delta, \phi)$	$=$	$\pi \circ (\delta \langle * \rangle, \phi)$
$\langle i \rangle \pi \circ (\delta, \phi)$	$=$	$\begin{cases} \pi \circ (\delta \langle \phi(i) \rangle, \phi), & i \in \mathrm{Dom}(\phi) \\ \pi \circ (\delta \langle * \rangle, \phi), & \text{otherwise} \end{cases}$
$\langle * \setminus i \rangle \pi \circ (\delta, \phi)$	$=$	$\begin{cases} \pi \circ (\delta \langle * \setminus \phi(i) \rangle, \phi), & i \in \mathrm{Dom}(\phi) \\ \pi \circ (\delta \langle * \rangle, \phi), & \text{otherwise} \end{cases}$

TABLE 6.8 – Deferred Paths – Application and Substitutions

Definition 6.6.3. Application of Symbolic Paths to a Dependency.

$$P \bullet (\delta, \phi) = \bigvee_{\forall \pi \in P} \pi \circ (\delta, \phi).$$

$\delta \blacktriangleleft (\sigma, \phi)$		
$\top \blacktriangleleft (\sigma, \phi)$	$=$	$\top$
$\oslash \blacktriangleleft (\sigma, \phi)$	$=$	$\oslash$
$\bot \blacktriangleleft (\sigma, \phi)$	$=$	$\bot$
$\{f_1 \mapsto \delta_1; \ldots; f_n \mapsto \delta_n\} \blacktriangleleft (\sigma, \phi)$	$=$	$\{f_1 \mapsto \delta_1 \blacktriangleleft (\sigma, \phi); \ldots; f_n \mapsto \delta_n \blacktriangleleft (\sigma, \phi)\}$
$[C_1 \mapsto \delta_1; \ldots; C_m \mapsto \delta_m] \blacktriangleleft (\sigma, \phi)$	$=$	$[C_1 \mapsto \delta_1 \blacktriangleleft (\sigma, \phi); \ldots; C_m \mapsto \delta_m \blacktriangleleft (\sigma, \phi)]$
$\textsf{Deferred}(\{o_1 \mapsto P_1; \ldots; o_k \mapsto P_k\}) \blacktriangleleft (\sigma, \phi)$	$=$	$\bigvee_{1 \le i \le k} P_i \bullet (\sigma(o_i), \phi)$
$\langle \delta_{def} \rangle \blacktriangleleft (\sigma, \phi)$	$=$	$\langle \delta_{def} \blacktriangleleft (\sigma, \phi) \rangle$
$\langle \delta_{def} \triangleright i \ : \ \delta_{exc} \rangle \blacktriangleleft (\sigma, \phi)$	$=$	$\begin{cases} \langle \delta_{def} \blacktriangleleft (\sigma, \phi) \triangleright \phi(i) : \ \delta_{exc} \blacktriangleleft (\sigma, \phi) \rangle & i \in \mathrm{Dom}(\phi) \\ \langle \delta_{def} \blacktriangleleft (\sigma, \phi) \vee \delta_{exc} \blacktriangleleft (\sigma, \phi) \rangle, & \text{otherwise} \end{cases}$

TABLE 6.9 – Interprocedural Domain – Substitutions

6.6.2 Wrapped Calls and Results

As a simple experiment for verifying the precision of our dependency analysis approach with deferred dependencies, we have replaced all calls to built-in predicates in our previous example predicates, `thread` and `start_address`, illustrated in Section 6.5.2 and on page 131, respectively, with calls to predicates wrapping every call of this type. We compared the precision of the obtained results, as well as the execution time needed to compute the dependency summaries.

The `thread_with_wrapped` predicate thus has the following form:

```
predicate thread_with_wrapped(process p, int i)
-> [true: thread ti|None|oob]
{{ array<option_thread> th, option_thread tio }} {
  get_threads(p)[true: th] : [true -> 1];
  get_ith(th, i)[true: tio|false] : [true -> 2, false -> 5];
  switch_option(tio)[none|some: ti] : [none -> 4, some -> 3];
  [true];
  [None];
  [oob]
}
```

The `start_address` predicate becomes:

```
predicate start_address_wrapped(process p, int j)
 -> [true: int adr|None]
{{thread tj, memory_region sj}} {
  thread(p, j)[true: tj | None | oob] : [true -> 1,
   None -> 4, oob -> 4];
  get_stack(tj) [true: sj] : [true -> 2];
  get_start(sj) [true: adr] : [true -> 3];
  [true];
  [None];
  [error]
}
```

The dependency summaries obtained for each of the two predicates are identical to the ones obtained for the predicates `thread` and `start_address` in their original form. The dependency information for `thread` and `start_address` is computed in 0.33 milliseconds, while that for the versions with calls to the wrapped built-in predicates, i.e. `thread_with_wrapped` and `start_address_wrapped` are obtained in 0.65 milliseconds. We ran the analysis 10001 times in a loop. The time measured includes only the execution of the analysis algorithms. It excludes the time required to load the input files, as well as the time spent printing the results.

6.7 Related Work

For the past few decades, interprocedural analyses have generated considerable interest in the static analysis community. They expand the scope of analysis beyond a procedure's limits in order to encompass the effect of callees on callers. The precision

of both data-flow and control-flow analyses is traditionally characterized in terms of context-sensitivity, i.e. computing information depending on the calling context, or, its dual, context-insensitivity. For control-flow analyses, the terms *polyvariant* and *monovariant* analyses are used interchangeably for the same distinction (Nielson and Nielson, 1999). In (Midtgaard, 2012), a comprehensive survey of control-flow analyses for functional programs is made. Context-sensitivity has the advantage of increased precision. However, the scalability of such analyses is frequently a major concern. The precision and performance impact of context-sensitivity is discussed by Lhoták and Hendren in (Lhoták and Hendren, 2006). In contrast, Ruf argues in (Ruf, 1995) that context-insensitivity leads to little or no precision penalty. Shapiro and Horwitz argue in (Shapiro and Horwitz, 1997) that using a more precise pointer analysis does in general lead to more precise results.

Sharir and Pnueli introduced in (Sharir and Pnueli, 1978) a comprehensive theory of interprocedural data-flow analyses for general frameworks. The first of them, the *functional approach* is based upon computing a context-sensitive summary of a function or procedure call. Procedures are viewed as collections of structured program blocks and input-output relations are established for each such block. Subsequently, the effect of procedure calls is computed by simply using such relations. The second approach proposed by Sharir and Pnueli is the *call-string approach*. Broadly speaking, this is based upon avoiding infeasible paths by matching corresponding calls and returns. It can be seen as an extension to intraprocedural data-flow analyses, in which only valid interprocedural paths are considered during graph traversal. This is achieved by tagging the propagated data with an encoded history of procedure calls, thus making the interprocedural flow explicit and increasing the accuracy of the propagated information. Both approaches are generic and can be used for a wide variety of analyses. Our form of interprocedural dependency analysis is closer to the functional approach. For each predicate of the analysed program, it computes a dependency summary as an input-output relation and then uses this summary whenever the predicate is called. Symbolic elements are used to allow callers to inject their own context information.

Though desirable in terms of precision, context-sensitivity is often considered prohibitively costly in terms of performance. In practice, many analyses make a compromise and relax to a certain degree this requirement for scalability. Our approach makes no exception either: it constitutes an application of context-sensitive information to summaries with deferred dependencies, which are essentially context-insensitive abstractions with context-sensitive leaves. Though not purely context-sensitive, we obtain a gain in precision without sacrificing scalability.

Purely context-sensitive analyses have been developed, especially in the area of points-to analyses (Gharat, Khedker, and Mycroft, 2016), but also for information flow control (Hammer and Snelting, 2009) or liveness analysis used for garbage collection (Asati et al., 2014). In (Khedker, Mycroft, and Rawat, 2011), Khedker et. al., present a lazy context-sensitive points-to analysis. Points-to information is computed only for the pointers that are live and the propagation of points-to information is sparse, being restricted to live ranges of pointers. Though our approach is not directly comparable to this approach, it is interesting to make a few general remarks. In (Khedker,

Mycroft, and Rawat, 2011), strong liveness is used for identifying the pointers that are directly used or which are used for defining pointers that are strongly live. On the other hand, we use strong dependency to identify and distinguish between input subelements that are directly needed for computing the output and input subelements that are simply copied into and forwarded as outputs. Thus, Khedker et. al. prevent the explosion of information by clearly distinguishing between relevant and irrelevant information. We achieve scalability by refining the notion of *needed* or *depending on*. Their analysis is fully context-sensitive and is based on the *call-string approach* (Sharir and Pnueli, 1978); our analysis shows a relaxed form of context-sensitivity and is closer to the *functional approach*.

Jensen et. al. present in (Jensen, Møller, and Thiemann, 2010) a technique based on lazy propagation for context-sensitive interprocedural analysis of JavaScript programs, i.e. programs with objects and first-class functions. Transfer functions may not be distributive, and hence the IFDS technique (Reps, Horwitz, and Sagiv, 1995; Padhye and Khedker, 2013) is not applicable. They propagate data-flow information "by need" in an iterative fixpoint algorithm.

The computation of relevant information is deferred in demand-driven analyses (Horwitz, Reps, and Sagiv, 1995; Heintze and Tardieu, 2001; Zheng and Rugina, 2008; Sridharan et al., 2005) as well. These compute the targeted results only at specific program points, thereby avoiding the effort of computing a global result. We compute dependency summaries with symbolic elements. These can be seen as dependency templates parameterized by a caller's context. Their instantiation is deferred and left to the callers.

6.8 Conclusion

We have presented an extension of our dependency analysis, introducing a relaxed form of context-sensitivity. Our solution is based on computing deferred dependencies consisting of symbolic access maps in which caller's can subsequently inject their specific context information on an as-needed basis. The dependency summaries for each predicate are computed only once. However, by including nested context-sensitive components at the summaries' leaves, we reduce the precision penalty exerted by our previous context-insensitive approach. The introduction of deferred dependencies required the introduction of an additional level of symbolic paths and path sets. However, the impact of this extension had a minimal impact on the dependency analysis at the intra- and interprocedural levels, imposing only the modification of the initialisation strategy and of the substitution operation. As we will discuss in Chapter 8, our extension of the dependency analysis with deferred dependencies led to an increase of 10%–20% in execution time on our used benchmark. However, it obtained more precise dependency information for 50% of the predicates included in the used benchmark.

Chapter 7

Correlation Analysis

> A thousand fibers connect us [...]; and among those fibers, as sympathetic threads, our actions run as causes, and they come back to us as effects.
>
> Hermann Melville

7.1 Introduction

In the field of Artificial Intelligence, the frame problem (McCarthy and Hayes, 1969) is loosely, but frequently described as "knowing what stays the same as actions occur in a changing world" (Morgenstern, 1995). In the realm of software verification, the frame problem refers to establishing the boundaries within which functions operate and it has notoriously tedious implications and consequences along two different axes: the specification of *frame properties* (Borgida, Mylopoulos, and Reiter, 1995) and their verification.

Another frequently used definition of the frame problem in the context of Artificial Intelligence refers to "*efficiently determining* what remains the same in a changing world" (Morgenstern, 1995). This definition is similar to the first, yet the initial words "efficiently determining" confer it a subtle, but crucial nuance. In this chapter we are rather interested in the latter and we address the issue of automatically detecting deep-state modifications in the context of αSmil, a functional language. In our "changing world", destructive updates are not allowed. The new state *out* of a structured value *in* is obtained by destructuring *in* and reconstructing it in *out* by copying unmodified subvalues from *in* and replacing in *out* only what needs to reflect the modification. Thus, referring to *old* values per se, as one of the three major approaches to specifying frame properties (described in Section 2.3.1) implies, does not make sense. Instead, we have to focus on, and to detect the relations between the (sub)values *in* and *out*. To this end we present a static *correlation* analysis which, when given a predicate that manipulates a structured input, is meant to determine automatically, the subset that remains unchanged and is further propagated into the output. Thus, the behaviour of a predicate is summarised by computing relations between parts of the input and parts of the output. The computed *correlation summaries* are a safe approximation of what

part of an input state of a predicate is copied to the output state; they summarise not only what is modified by the predicate but also *how* it is modified and *to what extent*.

Outline. We continue this chapter by illustrating the targeted correlation results on an αSmil example in Section 7.1.1. In Section 7.1.2 we give a brief overview of the characteristics of our correlation analysis and explain the motivation behind some of them. The rest of the chapter is focusing on technical details related to the correlation analysis. In Section 7.2 we present our abstract partial equivalence type, a fundamental component of our correlation analysis. It is followed in Section 7.3 by an in-depth presentation of *paths* and *correlations*, an intermediate level of abstraction that is imperative for obtaining expressive results. In Section 7.4, we focus on the correlation analysis at an intraprocedural level and illustrate the step-by-step mechanism behind it in Section 7.4.2. A summary of the correlation analysis at an interprocedural level is given in Section 7.5. A possible extension going beyond the detection of equivalences and handling more general relations is briefly discussed in Section 7.6. Detecting modifications is traditionally associated to shape and side-effect analyses. In Section 7.7 we review and discuss such approaches.

7.1.1 Targeted Correlation Information

The goal of our analysis and the targeted correlation results can be illustrated on an example predicate, such as `stop_thread`, for instance. This predicate has been introduced in Section 3.1.5 (on page 50) and its body in the αSmil language was shown in Section 4.1 on page 64. We revisit it and illustrate the predicate's body in Figure 7.1.

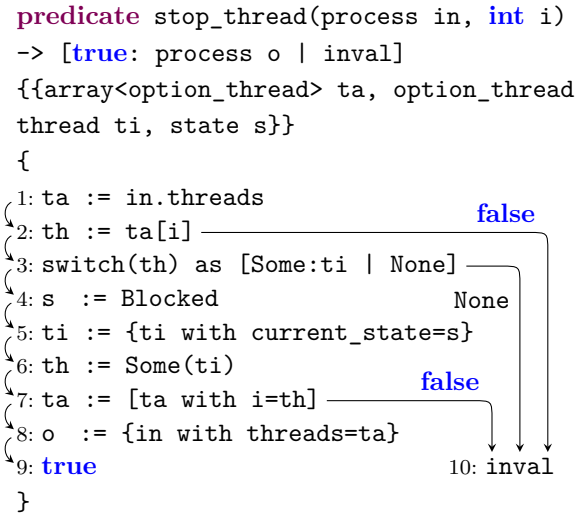


FIGURE 7.1 – Body of the `stop_thread` Predicate

It has two possible execution scenarios: **true**, when the given index `i` corresponds to an *active* thread, and `inval` otherwise, i.e. when it corresponds to an inactive element or when it lies outside the array's bounds. In the latter case, the predicate exits with

the `inval` label and generates no output. In the former case, `stop_thread` modifies the state of the `i`-th active thread by setting it to `Blocked`, and returns the new state of the process in the output `o`. This is accomplished by destructuring the input process `in` and copying the array of associated threads into the local variable `ta` (line 1). The array's `i`-th element is copied to the local variable `th` (line 2) and as it is an active element, its corresponding thread is extracted and put into `ti` (line 3). The new state for the thread value `ti` is created by setting its `current_state` field (line 5) to the state `s` constructed previously (line 4). The new state `o` of the process is constructed, using `ti` for its `i`-th active element (lines 6 and 7) and copying everything else from the input `in` (line 9). It is interesting to note that for each destructuring step of `in`, there is a corresponding construction step for `o`, as is visible at lines `1` and `8`, `2` and `7`, and `3` and `6`, for instance.

The targeted correlation results for this predicate are illustrated in Figure 7.2. Our analysis should infer that between the input process `in` and the output `o`, the values of the fields `pid`, `current_thread` and `address_space` are equal. Furthermore, for the `threads` array of associated threads it should detect that all elements are equal except the value of the `i`-th element (as illustrated by R_{th}), for which only one of the three fields, namely the `current_state` field, differs (shown by R_i).

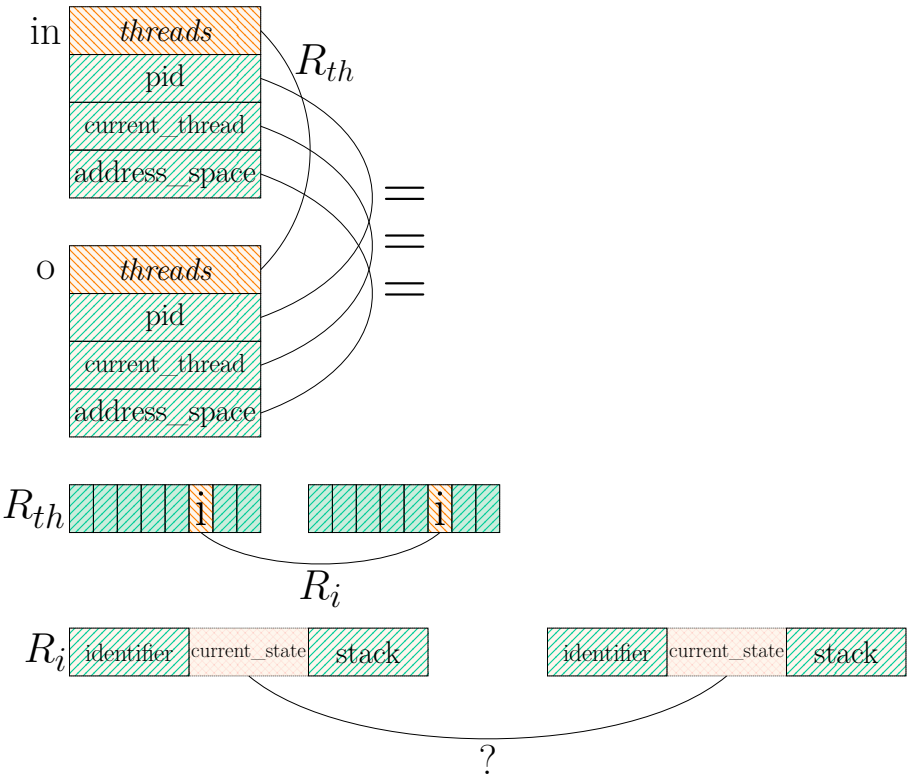


FIGURE 7.2 – Targeted Correlation Results for Predicate `stop_thread`

By tracking equalities between pairs of variables of the same type and by defining

an abstract partial equivalence type that mirrors the layered structure of associative arrays and algebraic data types, we can detect the equality of the values for the `pid`, `current_thread` and `address_space` fields between the input and the output. However, if we track *only* equalities between variables of the same type and we ignore the flow of an input's subelement value to a variable (or conversely, the flow of a variable's value to an output's subelement) valuable information is lost. We are not only losing information between inputs and outputs of different types, but by accumulating imprecisions, we also lose information concerning inputs and outputs of the same type, such as the `in` and `o` processes of our example. For instance, the equality between the values extracted from the input `in` and copied into `ta` and `th`, respectively, as well as the relation between the values of `ta` and `o.threads`, and `th` and `o.threads[i]` are ignored because neither `ta` nor `th` are of the same type as `in` and `o`. As a consequence, we lose the information concerning the relation between `in`'s and `o`'s `threads` values altogether. In order to compute such information it is imperative to track (cor)relations between variables of different types as well.

7.1.2 Correlation Analysis in a Nutshell

Our correlation analysis is a conservative static analysis inferring what is modified by an operation and to what extent. It approximates the flow of input values into output values, by uncovering *equalities* and computing *correlations* as pairs between input parts and the output parts into which these are injected. What is marked as being equal is definitely equal.

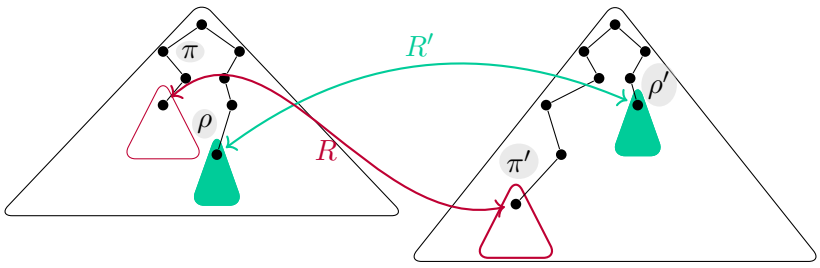
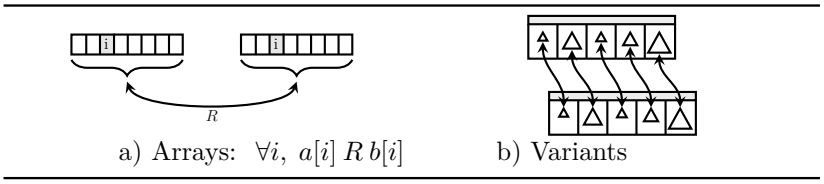


FIGURE 7.3 – Intraprocedural Correlations – General Representation

Outputs are often complex compounds of different subparts of different input variables: a subset of the input is modified, while the rest is injected as is. We track the origin of subparts of the output and relate it to subparts of the input. As previously illustrated on our `stop_thread` example predicate, in order to prevent avoidable over-approximations, we need to avoid dealing with data in a monolithic manner. To this end, it is imperative to consider pairs of different types and granularities as well. As a consequence, we are forced to introduce an additional level of granularity allowing us to refer not only to variables, but also to substructures within them. At the intraprocedural level, illustrated in Figure 7.3, we define correlations as mappings between pairs of inputs and outputs to which we associate mappings between pairs of valid inner paths

and the relations binding them. Correlations for arrays and variants are exemplified in Figures 7.4-a) and 7.4-b).



a) Arrays: $\forall i,\ a[i]\ R\ b[i]$ b) Variants

FIGURE 7.4 – Intraprocedural Domain – Examples

Similarly to our dependency analysis presented in Chapter 5, the correlation analysis is an interprocedural, flow-sensitive, field-sensitive, label-sensitive analysis that handles associative arrays, structures and variant data types. However, unlike the dependency analysis for which we introduced a relaxed form of context-sensitivity in Chapter 6, the correlation analysis is context-insensitive. Fine-grained equivalence relations between the inputs and outputs of a predicate are computed once and subsequently propagated to its callers.

Our correlation analysis is meant to be used in an interactive verification context. Precise correlation summaries must be computed quickly in order to answer effectively, when combined with dependency summaries, queries regarding the preservation of certain invariants.

7.2 Partial Equivalence Relations

7.2.1 Abstract Partial Equivalence Type

The first step towards automatically reasoning about the propagation of input subelements into output subelements is the definition of an abstract *partial equivalence type* $\mathscr{R}$ that mimics the structure of algebraic data types and arrays. A partial equivalence relation $R \in \mathscr{R}$ is defined inductively from the two atomic elements, Equal and Any, and mirrors the structure of the concrete types.

Definition 7.2.1. Partial Equivalence Type $R \in \mathscr{R}$.

$R :=$	$\mid$ Equal	atomic case – equal	(i)
	$\mid$ Any	atomic case – unrelated	(ii)
	$\mid \{f_1 \mapsto R_1; \ldots; f_n \mapsto R_n\}$	$f_1, \ldots, f_n$ fields	(iii)
	$\mid [C_1 \mapsto R_1; \ldots; C_n \mapsto R_n]$	$C_1, \ldots, C_n$ constructors	(iv)
	$\mid \langle R_{def} \rangle$	array	(v)
	$\mid \langle R_{def} \triangleright i \ :\ R_{exc} \rangle$	i array index	(vi)

Such relations represent fine-grained partial equivalences between pairs of values of the same type. Equal and Any represent equal and unrelated values, respectively. Partial equivalence relations for structures (given by (iii)) and for variants (given by (iv)), are expressed in terms of the partial equivalences of their subparts, by mapping each field

or constructor to the corresponding relations. As for the dependency analysis presented in Chapter 5, for arrays, we distinguish between two cases, namely arrays with a general relation applying to all of the cells (as given by (v)) or to all but one exceptional cell (as given by (vi)), for which a specific relation is known to hold.

The preorder relation of the partial equivalence lattice is denoted by $\sqsubseteq_{\mathscr{R}}$ and defined below.

Definition 7.2.2. Preorder Relation $\sqsubseteq_{\mathscr{R}}$.

$$\sqsubseteq_{\mathscr{R}} \subseteq \mathscr{R} \times \mathscr{R}.$$

It is detailed in Table 7.1.

TABLE 7.1 – $\sqsubseteq_{\mathscr{R}}$ – Comparison of Two Domains

$$\frac{}{R \sqsubseteq_{\mathscr{R}} \mathsf{Any}}\ \textsc{Top} \qquad \frac{}{\mathsf{Equal} \sqsubseteq_{\mathscr{R}} R}\ \textsc{Bot}$$

$$\frac{R_1 \sqsubseteq_{\mathscr{R}} R'_1 \quad \dots \quad R_n \sqsubseteq_{\mathscr{R}} R'_n}{\{f_1 \mapsto R_1; \dots; f_n \mapsto R_n\} \sqsubseteq_{\mathscr{R}} \{f_1 \mapsto R'_1; \dots; f_n \mapsto R'_n\}}\ \textsc{Str}$$

$$\frac{R_1 \sqsubseteq_{\mathscr{R}} R'_1 \quad \dots \quad R_n \sqsubseteq_{\mathscr{R}} R'_n}{[C_1 \mapsto R_1; \dots; C_n \mapsto R_n] \sqsubseteq_{\mathscr{R}} [C_1 \mapsto R'_1; \dots; C_n \mapsto R'_n]}\ \textsc{Var}$$

$$\frac{R \sqsubseteq_{\mathscr{R}} R'}{\langle R \rangle \sqsubseteq_{\mathscr{R}} \langle R' \rangle}\ \textsc{Adef} \qquad \frac{R_{def} \sqsubseteq_{\mathscr{R}} R'_{def} \quad R_{exc} \sqsubseteq_{\mathscr{R}} R'_{exc}}{\langle R_{def} \triangleright i \ :\ R_{exc} \rangle \sqsubseteq_{\mathscr{R}} \left\langle R'_{def} \triangleright i \ :\ R'_{exc} \right\rangle}\ \text{AI}$$

$$\frac{R_{def} \sqsubseteq_{\mathscr{R}} R' \quad R_{exc} \sqsubseteq_{\mathscr{R}} R'}{\langle R_{def} \triangleright i \ :\ R_{exc} \rangle \sqsubseteq_{\mathscr{R}} \langle R' \rangle}\ \text{AIA} \qquad \frac{R \sqsubseteq_{\mathscr{R}} R'_{def} \quad R \sqsubseteq_{\mathscr{R}} R'_{exc}}{\langle R \rangle \sqsubseteq_{\mathscr{R}} \left\langle R'_{def} \triangleright i \ :\ R'_{exc} \right\rangle}\ \text{AAI}$$

$$\frac{i \neq j \quad R_{def} \sqsubseteq_{\mathscr{R}} R'_{def} \quad R_{def} \sqsubseteq_{\mathscr{R}} R'_{exc} \quad R_{exc} \sqsubseteq_{\mathscr{R}} R'_{def} \quad R_{exc} \sqsubseteq_{\mathscr{R}} R'_{exc}}{\langle R_{def} \triangleright i \ :\ R_{exc} \rangle \sqsubseteq_{\mathscr{R}} \left\langle R'_{def} \triangleright j \ :\ R'_{exc} \right\rangle}\ \text{AIJ}$$

The *join* and *meet* operations are denoted by $\vee_{\mathscr{R}}$ and $\wedge_{\mathscr{R}}$, respectively.

Definition 7.2.3. Join Operation $\vee_{\mathscr{R}}$.

$$\vee_{\mathscr{R}} : \mathscr{R} \times \mathscr{R} \to \mathscr{R}.$$

Definition 7.2.4. Meet Operation $\wedge_{\mathscr{R}}$.

$$\wedge_{\mathscr{R}} : \mathscr{R} \times \mathscr{R} \to \mathscr{R}.$$

Both are *commutative* operations, applied pointwise on each subelement. *Join*, shown in Table 7.2, has Equal as its identity element and Any as its absorbing element. *Meet*, shown in Table 7.3, has Equal as its absorbing element and Any as its identity element. For both operations the undisplayed cases are defined by their symmetrical counterparts.

TABLE 7.2 – Partial Equivalences – $\vee_{\mathscr{R}}$ – Join Operation

R'		R''		$R' \vee_{\mathscr{R}} R''$
Any	$\vee_{\mathscr{R}}$	R	$=$	Any
Equal	$\vee_{\mathscr{R}}$	R	$=$	R
$\{f_1 \mapsto R_1; \ldots; f_n \mapsto R_n\}$	$\vee_{\mathscr{R}}$	$\{f_1 \mapsto R'_1; \ldots; f_n \mapsto R'_n\}$	$=$	$\{f_1 \mapsto R_1 \vee_{\mathscr{R}} R'_1; \ldots; f_n \mapsto R_n \vee_{\mathscr{R}} R'_n\}$
$[C_1 \mapsto R_1; \ldots; C_n \mapsto R_n]$	$\vee_{\mathscr{R}}$	$[C_1 \mapsto R'_1; \ldots; C_n \mapsto R'_n]$	$=$	$[C_1 \mapsto R_1 \vee_{\mathscr{R}} R'_1; \ldots; C_n \mapsto R_n \vee_{\mathscr{R}} R'_n]$
$\langle R \rangle$	$\vee_{\mathscr{R}}$	$\langle R' \rangle$	$=$	$\langle R \vee_{\mathscr{R}} R' \rangle$
$\langle R \rangle$	$\vee_{\mathscr{R}}$	$\langle R'_{def} \triangleright i : R'_{exc} \rangle$	$=$	$\langle R \vee_{\mathscr{R}} R'_{def} \triangleright i : R \vee_{\mathscr{R}} R'_{exc} \rangle$
$\langle R_{def} \triangleright i : R_{exc} \rangle$	$\vee_{\mathscr{R}}$	$\langle R'_{def} \triangleright j : R'_{exc} \rangle \begin{cases} i = j \\ i \neq j \end{cases}$	$=$	$\langle R_{def} \vee_{\mathscr{R}} R'_{def} \triangleright i : R_{exc} \vee_{\mathscr{R}} R'_{exc} \rangle$ $\langle R_{def} \vee_{\mathscr{R}} R'_{def} \vee_{\mathscr{R}} R_{exc} \vee_{\mathscr{R}} R'_{exc} \rangle$

TABLE 7.3 – Partial Equivalences – $\wedge_{\mathscr{R}}$ – Meet Operation

R'		R''		$R' \wedge_{\mathscr{R}} R''$
Any	$\wedge_{\mathscr{R}}$	R	$=$	R
Equal	$\wedge_{\mathscr{R}}$	R	$=$	Equal
$\{f_1 \mapsto R_1; \ldots; f_n \mapsto R_n\}$	$\wedge_{\mathscr{R}}$	$\{f_1 \mapsto R'_1; \ldots; f_n \mapsto R'_n\}$	$=$	$\{f_1 \mapsto R_1 \wedge_{\mathscr{R}} R'_1; \ldots; f_n \mapsto R_n \wedge_{\mathscr{R}} R'_n\}$
$[C_1 \mapsto R_1; \ldots; C_n \mapsto R_n]$	$\wedge_{\mathscr{R}}$	$[C_1 \mapsto R'_1; \ldots; C_n \mapsto R'_n]$	$=$	$[C_1 \mapsto R_1 \wedge_{\mathscr{R}} R'_1; \ldots; C_n \mapsto R_n \wedge_{\mathscr{R}} R'_n]$
$\langle R \rangle$	$\wedge_{\mathscr{R}}$	$\langle R' \rangle$	$=$	$\langle R \wedge_{\mathscr{R}} R' \rangle$
$\langle R \rangle$	$\wedge_{\mathscr{R}}$	$\langle R'_{def} \triangleright i : R'_{exc} \rangle$	$=$	$\langle R \wedge_{\mathscr{R}} R'_{def} \triangleright i : R \wedge_{\mathscr{R}} R'_{exc} \rangle$
$\langle R_{def} \triangleright i : R_{exc} \rangle$	$\wedge_{\mathscr{R}}$	$\langle R'_{def} \triangleright j : R'_{exc} \rangle \begin{cases} i = j \\ i \neq j \end{cases}$	$=$	$\langle R_{def} \wedge_{\mathscr{R}} R'_{def} \triangleright i : R_{exc} \wedge_{\mathscr{R}} R'_{exc} \rangle$ $\langle R_{def} \wedge_{\mathscr{R}} R'_{def} \wedge_{\mathscr{R}} R_{exc} \wedge_{\mathscr{R}} R'_{exc} \rangle$

Additionally, extraction functions are defined for partial equivalence relations.

Definition 7.2.5. Extraction of a Field's Relation:

$$\mathit{extr}_f \ : \ \mathscr{R} \ \nrightarrow \ \mathscr{R}.$$

Definition 7.2.6. Extraction of a Constructor's Relation:

$$\mathit{extr}_C \ : \ \mathscr{R} \ \nrightarrow \ \mathscr{R}.$$

Definition 7.2.7. Extraction of a Cell's Relation:

$$\mathit{extr}_{\langle i \rangle} \ : \ \mathscr{R} \ \nrightarrow \ \mathscr{R}.$$

These are partial functions and can only be applied on relations of the corresponding types. For example, the field extraction extr_f only makes sense for atomic or structured relations having a field named f, which should be the case if the relation connects two values of a structured type with a field f. For any of the two atomic relations Equal or Any, applying any of these extractions yields Equal or Any, respectively. They are summarized in Table 7.4.

TABLE 7.4 – Partial Equivalence Extractions

$\mathit{extr}_f(R), f \in \mathcal{F}$		
$\mathit{extr}_f(\mathsf{Any})$	$=$	Any
$\mathit{extr}_f(\mathsf{Equal})$	$=$	Equal
$\mathit{extr}_f(\{f_1 \mapsto R_1; \ldots; f_i \mapsto R_i; \ldots; f_n \mapsto R_n\})$	$=$	R_i if $f = f_i$

$\mathit{extr}_C(R), C \in \mathcal{C}$		
$\mathit{extr}_C(\mathsf{Any})$	$=$	Any
$\mathit{extr}_C(\mathsf{Equal})$	$=$	Equal
$\mathit{extr}_C([C_1 \mapsto R_1; \ldots; C_i \mapsto R_i; \ldots; C_n \mapsto R_n])$	$=$	R_j if $C = C_j$

$\mathit{extr}_{\langle i \rangle}(R)$		
$\mathit{extr}_{\langle i \rangle}(\mathsf{Any})$	$=$	Any
$\mathit{extr}_{\langle i \rangle}(\mathsf{Equal})$	$=$	Equal
$\mathit{extr}_{\langle i \rangle}(\langle R \rangle)$	$=$	R
$\mathit{extr}_{\langle i \rangle}(\langle R_{def} \triangleright i \ : \ R_{exc} \rangle)$	$=$	R_{exc}
$\mathit{extr}_{\langle i \rangle}(\langle R_{def} \triangleright j \ : \ R_{exc} \rangle), i \neq j$	$=$	$R_{def} \vee_{\mathscr{R}} R_{exc}$

7.2.2 Well-Typed Partial Equivalences and their Semantics

As discussed in the case of dependencies in Section 5.2.2, syntactic partial equivalences are untyped. However, their interpretation is made in the context of a type $\tau \in \mathbb{T}$. The atomic cases, such as Equal and Any can apply to any type since they are not exhibiting any data type features. Cases other than Equal and Any only have non-empty interpretations for types τ which are compatible with their shape. For instance, the structured relation $\{f \mapsto R\}$ only really makes sense for structured types with a single field f, whose type itself is compatible with R, and will not be used in connection with variant or array types for example. In Table 7.5 we detail the inference rules related to the *well-typedness* of partial equivalences. This is described as a judgement parameterized by a typing environment Γ (Definition 4.3.1).

$$\frac{}{\Gamma \vdash \mathsf{Equal} \ : \ \tau}\ \mathrm{WT}\top \qquad \frac{}{\Gamma \vdash \mathsf{Any} \ : \ \tau}\ \mathrm{WT}\bot$$

$$\frac{\tau = \textbf{\textit{struct}}\{f_1 : \tau_1, \ldots, f_n : \tau_n\} \quad \Gamma \vdash R_1 : \tau_1 \quad \ldots \quad \Gamma \vdash R_n : \tau_n}{\Gamma \vdash \{f_1 \mapsto R_1; \ldots; f_n \mapsto R_n\} \ : \ \tau} \ \text{WTStruct}$$

$$\frac{\tau = \textbf{\textit{variant}}[C_1 : \tau_1 | \ \ldots | \ C_n : \tau_n] \quad \Gamma \vdash R_1 : \tau_1 \quad \ldots \quad \Gamma \vdash R_n : \tau_n}{\Gamma \vdash [C_1 \mapsto R_1; \ldots; C_n \mapsto R_n] \ : \ \tau} \ \text{WTVar}$$

$$\frac{\Gamma \vdash R \ : \ \tau}{\Gamma \vdash \langle R \rangle \ : \ \textbf{\textit{arr}}^{\tau_i}\langle \tau \rangle} \ \text{WTArr}$$

$$\frac{\Gamma \vdash R_{def} \ : \ \tau \quad \Gamma \vdash R_{exc} \ : \ \tau \quad \Gamma(i) = \tau_i}{\Gamma \vdash \langle R_{def} \triangleright i \ : \ R_{exc} \rangle \ : \ \textbf{\textit{arr}}^{\tau_i}\langle \tau \rangle} \ \text{WTArrI}$$

Table 7.5 – Well-Typed Partial Equivalences

The atomic values are *generic*: they are well-typed with respect to any type (WT$\top$, WT$\bot$). The partial equivalences of structures (WTStruct) are well-typed only with respect to an adequate structured type, whose field types are themselves compatible with the equivalences mapped to them. Similarly, the partial equivalences of variants (WTVar) are well-typed only with respect to an adequate variant type. In turn, the constructors must be themselves pointwise compatible with the equivalences mapped to them. For well-typed array equivalences (WTArr, WTArrI), the default relation as well as the exceptional relation have to be compatible with the type τ of the array's elements. Furthermore, the type of i, the index of the known exceptional equivalence relation, has to be compatible with τ_i, the array's index type.

The semantics of a partial equivalence R for a type τ is a partial equivalence relation over values of type τ. Given a *valuation* E from variables to semantic values (Definition 4.4.2), the interpretation $[\![R]\!]^\tau$ of a relation $R \in \mathscr{R}$ with respect to some type τ is a binary relation over $\mathbb{D}_\tau$ (Definition 4.4.1). The interpretation $[\![R]\!]^\tau$ is defined as shown in Table 7.6.

$$[\![\mathsf{Equal}]\!]^\tau = \{(x, x) |\ x \in \mathbb{D}_\tau\} \qquad [\![\mathsf{Any}]\!]^\tau = \mathbb{D}_\tau \times \mathbb{D}_\tau$$

$$[\![\{f_1 \mapsto R_1; \ldots; f_n \mapsto R_n\}]\!]^{\textbf{\textit{struct}}\{f_1:\tau_1,\ldots,f_n:\tau_n\}} = \{(\{f_1 = v_1, \ldots, f_n = v_n\}, \{f_1 = w_1, \ldots, f_n = w_n\}) \mid \forall i, 1 \leq i \leq n, \ (v_i, w_i) \in [\![R_i]\!]^{\tau_i}\}$$

$$[\![[C_1 \mapsto R_1; \ldots; C_n \mapsto R_n]]\!]^{\textbf{\textit{variant}}[C_1:\tau_1 | \ldots | C_n:\tau_n]} = \{(C_i[v_i], C_i[w_i]) \mid \ (v_i, w_i) \in [\![R_i]\!]^{\tau_i}\}$$

$$[\![\langle R_{def} \rangle]\!]^{\textbf{\textit{arr}}^{\tau_i}\langle \tau \rangle} = \{((\mathcal{P}, (v)_k), (\mathcal{P}, (w)_k)) \mid \ \forall k, (v_k, w_k) \in [\![R_{def}]\!]^\tau\}$$

$$[\![\langle R_{def} \triangleright i \ : \ R_{exc} \rangle]\!]^{\mathbf{arr}^{\tau_i}\langle\tau\rangle} = \{((\mathcal{P},(v)_k),(\mathcal{P},(w)_k)) \mid E(i) \in \mathcal{P} \Longrightarrow (v_{E(i)}, w_{E(i)}) \in [\![R_{exc}]\!]^{\tau},\ \forall k \neq E(i), (v_k, w_k) \in [\![R_{def}]\!]^{\tau}\}$$

TABLE 7.6 – Partial Equivalence Relations – Semantics

A partial equivalence relation R only relates values of the same type τ, which must be compatible with R's "shape". For structures, a partial equivalence relates pointwise the values of the fields of the two structure values. For variant values, a partial equivalence relation relates values built with the same constructor C_i, using arguments whose values are related by a relation R_i. For arrays, $\mathcal{P}$ indicates the support type, which has to be identical for both values. The values of the array elements are pointwise related by the same relation R_{def}, with the exception of the i-th elements which are potentially related by an exceptional relation R_{exc}. Since variables i are used for indicating the exceptional elements, the valuation E is used for determining the value of i.

7.3 Paths and Correlations

7.3.1 Paths and Correlation Types

The partial equivalence relations discussed in Section 7.2 and defined in 7.2.1 are enough to represent fine-grained information for values of the same structured type. For the `stop_thread` example discussed in Section 7.1.1 these would suffice to express the equality of the `pid`, `current_thread` and `address_space` fields between the input process `in` and the output process `o`, by simply mapping this pair to the following partial equivalence:

$$\left\{ \begin{array}{lcl} \texttt{threads} & \mapsto & \mathsf{Any} \\ \texttt{pid} & \mapsto & \mathsf{Equal} \\ \texttt{current\_thread} & \mapsto & \mathsf{Equal} \\ \texttt{address\_space} & \mapsto & \mathsf{Equal} \end{array} \right\}.$$

However, the partial equivalence relations cannot, for instance, be used to convey the equality at line 1 in Figure 7.1 between the value of the `threads` field of `in` and the local `ta` variable. By not tracking information such as this, we lose the targeted information regarding the `threads` field, denoted by R_{th} in Figure 7.2. In order to express this information, we first need to be able to refer to the substructure `in.threads` and relate its value to the one of `ta`.

To this end, rather than handling only partial equivalences between pairs of variables of the same type and approximating the rest to Any – the element that conveys no information – we introduce an intermediate level, allowing us to store relations between subparts of values. We begin by introducing *access paths*. Unlike the *symbolic paths* introduced in Chapter 6 and defined in 6.3.1, that are used for computing dependency summaries with context-sensitive elements, the paths used for the correlation analysis

are *actual access* paths inside some value's structure. The symbolic paths used in deferred dependencies may cover multiple actual paths inside a value, whereas the *access paths* required for the correlation analysis represent unique chains of internal accesses leading to a *single*, nested subvalue. Each access path is rooted at one of the program's variables. It is noteworthy to remark that in both cases, an intermediate level below variables needs to be introduced as soon as fine-grained relations between pairs of variables are considered, directly or indirectly. In the case of deferred dependencies this was not the main goal per se but rather a mechanism for obtaining more precision in specific cases for already pertinent dependency results. In contrast, for the correlation results this is imperative for obtaining useful, expressive information in non-trivial cases. We therefore define a recursive type $\widehat{\pi} \in \widehat{\Pi}$ encompassing this.

Definition 7.3.1. Access Path Type $\widehat{\pi} \in \widehat{\Pi}$.

$$\begin{array}{rll} \widehat{\pi} := & \mid \widehat{\varepsilon} & \text{empty – root} \\ & \mid .f\widehat{\pi} & f \in \mathcal{F} \\ & \mid @C\widehat{\pi} & C \in \mathcal{C} \\ & \mid \langle i \rangle \widehat{\pi} & i \text{ index, program variable.} \end{array}$$

The *empty* path, denoted by $\widehat{\varepsilon}$, is the special case denoting an access to an entire element, i.e. the root. The action of appending a *non-empty* path $\widehat{\pi}'$ to another path $\widehat{\pi}$ is denoted by $\widehat{\pi} :: \widehat{\pi}'$. For instance, the path denoting the `current_state` field of the `i`-th active, associated thread of the `in` process of our `stop_thread` predicate would be the following: *in.threads⟨i⟩@Some.t.current_thread*.

Meaningful information is conveyed by associating paths and partial equivalence relations. For instance, the equality between `in.threads` and `ta` at line 1 in Figure 7.1 can be expressed by associating Equal to the pair of subelements identified by the `.threads` path in `in` and by $\widehat{\varepsilon}$ in `ta`. We call *correlation* such a mapping from a pair of access paths to a partial relation. After setting the `i`-th element of `ta` to `ti`, the thread with the current state set to `Blocked` and everything else left unmodified, we could express the relation between `in` and `ta` by two correlations, namely:

$$\begin{array}{rcl} (\texttt{.threads}, \widehat{\varepsilon}) & \mapsto & \langle \mathsf{Equal} \triangleright \texttt{i} : \mathsf{Any} \rangle \\ (\texttt{.threads}\langle \texttt{i} \rangle @\texttt{Some.t}, \langle \texttt{i} \rangle @\texttt{Some.t}) & \mapsto & \left\{ \begin{array}{rcl} \texttt{identifier} & \mapsto & \mathsf{Equal} \\ \texttt{current\_state} & \mapsto & \mathsf{Any} \\ \texttt{stack} & \mapsto & \mathsf{Equal} \end{array} \right\}. \end{array}$$

To this end, we introduce *correlation maps* $\kappa \in \mathscr{K}$, defined below.

Definition 7.3.2. Correlation Maps $\kappa \in \mathscr{K}$.

Correlation maps $\kappa \in \mathscr{K}$ are finite mappings from pairs of paths to partial equivalence relations $R \in \mathscr{R}$:

$$\kappa : \widehat{\Pi} \times \widehat{\Pi} \rightarrow \mathscr{R}.$$

Generally, for two given variables e and o, a correlation $(\widehat{\pi}, \widehat{\rho}) \mapsto R$ specifies that e and o have nested subelements, respectively identified by the inner paths $\widehat{\pi}$ and $\widehat{\rho}$, whose values are related by the relation R.

We conclude this subsection by specifying what it means for paths, correlations and correlation maps to be *well-typed*.

For characterizing the contexts in which an access path $\widehat{\pi}$ is *well-typed*, we need to consider the types of values to which it can be applied and the types of (sub)values to which it can lead to. Therefore, in the following, we define a typing judgement for access paths as a three-place relation $\widehat{\pi} : \tau \to \tau'$, whose meaning is that $\widehat{\pi}$ can be applied to any value of type τ and in that case it will always describe subvalues of type τ'. Additionally, the typing judgement is also parameterized by a set of *input variables* $\mathcal{I}$, which are the variables having the right to appear as identifiers for array accesses. This is detailed in Table 7.7.

$$\frac{}{\Gamma, \mathcal{I} \vdash \widehat{\varepsilon} \ : \ \tau \ \to \ \tau} \ \text{WT}\widehat{\varepsilon}$$

$$\frac{\tau = \textbf{\textit{struct}}\{f_1 : \tau_1, \ldots, f_i : \tau_i, \ldots, f_n : \tau_n\} \quad \Gamma, \mathcal{I} \vdash \widehat{\pi}_i : \tau_i \to \tau'}{\Gamma, \mathcal{I} \vdash .f_i\widehat{\pi}_i \ : \ \tau \ \to \ \tau'} \ \text{WTStructAPath}$$

$$\frac{\tau = \textbf{\textit{variant}}[C_1 : \tau_1 | \ \ldots | \ C_i : \tau_i | \ \ldots | \ C_n : \tau_n] \quad \Gamma, \mathcal{I} \vdash \widehat{\pi}_i : \tau_i \to \tau'}{\Gamma, \mathcal{I} \vdash @C_i\widehat{\pi}_i \ : \ \tau \ \to \ \tau'} \ \text{WTVarAPath}$$

$$\frac{\Gamma, \mathcal{I} \vdash \widehat{\pi}_i \ : \ \tau \ \to \ \tau' \qquad \Gamma(i) = \tau_i \qquad i \in \mathcal{I}}{\Gamma, \mathcal{I} \vdash \langle i \rangle \widehat{\pi}_i \ : \ \textbf{\textit{arr}}^{\tau_i}\langle \tau \rangle \ \to \ \tau'} \ \text{WTCellAPath}$$

Table 7.7 – Well-Typed Access Paths

Correlations are mappings from pairs of access paths to partial relations. Though the two access paths can be applied to values of different types, they both need to return subvalues of the same type τ'. Furthermore, the partial equivalence relation associated to them, has to be well-typed with respect to τ', as detailed in Table 7.5. The inference rule for well-typed correlations is shown in Table 7.8.

$$\frac{\Gamma, \mathcal{I} \vdash \widehat{\pi} \ : \ \tau_l \ \to \ \tau' \qquad \Gamma, \mathcal{I} \vdash \widehat{\rho} \ : \ \tau_r \ \to \ \tau' \qquad \Gamma \vdash R \ : \ \tau'}{\Gamma, \mathcal{I} \vdash (\widehat{\pi}, \widehat{\rho}) \mapsto R \ : \ (\tau_l, \tau_r)} \ \text{WTCorrelation}$$

Table 7.8 – Well-Typed Correlations

Finally, as shown in Table 7.9, a correlation map κ is well-typed if all the correlations it contains are well-typed.

$$\frac{\forall(\widehat{\pi},\widehat{\rho}) \mapsto R \in \kappa,\ \Gamma,\mathcal{I} \vdash (\widehat{\pi},\widehat{\rho}) \mapsto R\ :\ (\tau_l,\tau_r)}{\Gamma,\mathcal{I} \vdash \kappa\ :\ (\tau_l,\tau_r)}\ \textsc{WTCorMaps}$$

TABLE 7.9 – Well-Typed Correlation Maps

7.3.2 Alignment and Partial Order

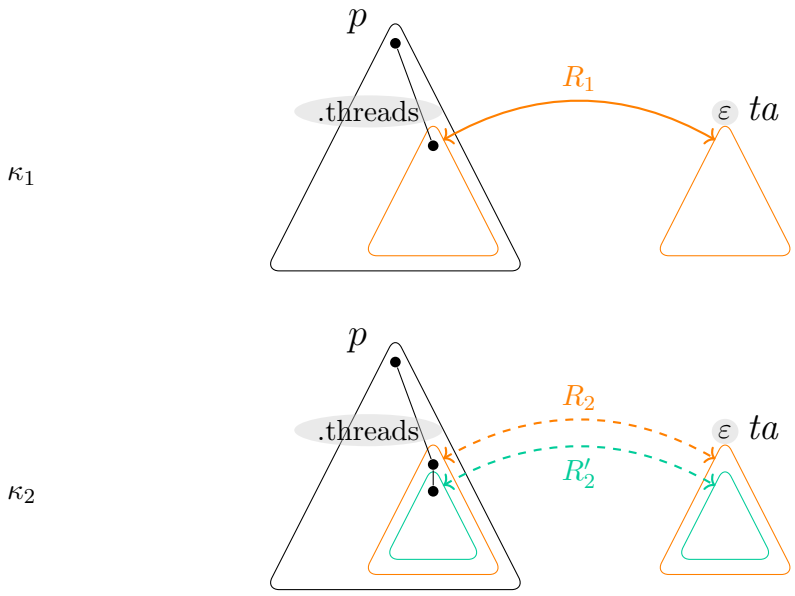
There is no clear choice for a canonical form for correlations. For instance, it is equivalent to write $(\widehat{\varepsilon},\widehat{\varepsilon}) \mapsto \{f \mapsto R\}$ and $(.f,.f) \mapsto R$. Is one superior to the other? Which one should be chosen? Operations can create and manipulate correlations in different manners, that are hard to predict. New correlations can also be introduced while considering *def-use* chains in the transfer function presented later in Section 7.4.1. Choosing between the two forms considerably limits flexibility. Not choosing a canonical form however has consequences as well; notably, it renders the definition of a partial order between correlation maps difficult. In order to compare two correlation maps κ_1 and κ_2, we cannot simply verify if the path pairs are identical and compare their associated relations. A correlation of the second map could be *linked*, in different manners, to multiple mappings of the first.

For instance, between a process `p` of the type used by our `stop_thread` example and an array `ta` of the same type as the field `threads` of the process, we might have the following correlation maps:

$$\kappa_1 :\quad (.\texttt{threads},\widehat{\varepsilon}) \mapsto \left\langle \begin{bmatrix} \texttt{None} & \mapsto & \mathsf{Any} \\ \texttt{Some} & \mapsto & \left\{ \texttt{t} \mapsto \left\{ \begin{array}{rcl} \texttt{identifier} & \mapsto & \mathsf{Equal} \\ \texttt{current\_state} & \mapsto & \mathsf{Any} \\ \texttt{stack} & \mapsto & \mathsf{Equal} \end{array} \right\} \right\} \end{bmatrix} \right\rangle$$

$$\kappa_2 :\quad \begin{array}{l} (.\texttt{threads},\widehat{\varepsilon}) \mapsto \langle \mathsf{Equal} \triangleright \texttt{i} :\ \mathsf{Any} \rangle \\ (.\texttt{threads}\langle\texttt{i}\rangle@\texttt{Some.t}, \langle\texttt{i}\rangle@\texttt{Some.t}) \mapsto \left\{ \begin{array}{rcl} \texttt{identifier} & \mapsto & \mathsf{Equal} \\ \texttt{current\_state} & \mapsto & \mathsf{Any} \\ \texttt{stack} & \mapsto & \mathsf{Equal} \end{array} \right\}. \end{array}$$

These correlation maps can be depicted as follows:



As illustrated above, in the given example map κ_2, in addition to the relation R_2 associated to $(\texttt{.threads}, \widehat{\varepsilon})$, the relation associated to $(\texttt{.threads}\langle\texttt{i}\rangle@\texttt{Some.t}, \langle\texttt{i}\rangle@\texttt{Some.t})$ and denoted by R_2', expresses information about the values of the process' `threads` field and `ta` as well. These are nested in the `i`-th element of each, as identified by $\langle\texttt{i}\rangle@\texttt{Some.t}$. In order to compare these two correlation maps, we have to first determine the relationships between the pair of paths $(\texttt{.threads}, \widehat{\varepsilon})$ from κ_1 and each pair of paths of κ_2. The first pair of paths in κ_2 is identical, whereas the second pair refers to elements that are further away from the root. Based on these relationships, we have to extract all the information relevant to $(\texttt{.threads}, \widehat{\varepsilon})$ from κ_2 and consider it in its entirety. This amounts to:

$$(\texttt{.threads}, \widehat{\varepsilon}) \mapsto \left\langle \mathsf{Equal} \triangleright \texttt{i} : \left[\begin{array}{lll} \texttt{None} & \mapsto & \mathsf{Any} \\ \texttt{Some} & \mapsto & \left\{ \texttt{t} \mapsto \left\{ \begin{array}{rll} \texttt{identifier} & \mapsto & \mathsf{Equal} \\ \texttt{current\_state} & \mapsto & \mathsf{Any} \\ \texttt{stack} & \mapsto & \mathsf{Equal} \end{array} \right\} \right\} \end{array} \right] \right\rangle .$$

Having expressed the information from the κ_2 correlation map at the same level as the information of κ_1 is expressed, i.e. that of the pair of paths $(\texttt{.threads}, \widehat{\varepsilon})$, we can finally compare them and conclude that the information contained by κ_2 is more precise than the relation associated to $(\texttt{.threads}, \widehat{\varepsilon})$ in κ_1. The relation associated to $(\texttt{.threads}, \widehat{\varepsilon})$ in κ_1 captures the equality between the values of the `identifier` and `stack` fields of *all active thread elements* of the two arrays identified by the paths. The relation associated to $(\texttt{.threads}, \widehat{\varepsilon})$ in κ_2 expresses the equality between *all* thread elements of the two arrays, except the `i`-th elements. Furthermore, if the `i`-th elements of the two arrays are *active*, it captures the equality between the values of the `identifier` and `stack` fields. Thus, by using the information contained by κ_1 we can conclude that for

all active elements of the two arrays, the values of 2 out of the 3 fields are equal; by using the more precise information contained by κ_2 we can conclude that all elements of the two arrays are equal, except the `i`-th one, for which the values of the same 2 out of 3 fields as in κ_1 are equal.

In the general case, for comparing two correlation maps κ_1 and κ_2, we need to collect for each correlation $(\widehat{\pi}, \widehat{\rho}) \mapsto R$ in κ_2 all the information contained by κ_1 that refers to the elements identified by $(\widehat{\pi}, \widehat{\rho})$ and verify if this covers at least the same information as the relation R. This information could be scattered across multiple mappings of the correlation map κ_1. We call *alignment* the process of collecting for any correlation $(\widehat{\pi}, \widehat{\rho}) \mapsto R$ in κ_2 all the information contained in κ_1 that refers to the elements identified by $(\widehat{\pi}, \widehat{\rho})$. It is necessary in the absence of a canonical form, a trait of our approach that is both a weakness and a strength: it leads to complex computations but gives considerable flexibility, as will be shown in Section 7.4.

For aligning, we first determine the relationships between paths by determining the relationship between the sequences of internal accesses that they represent. These can be identical, representing the same traversal to the same subelement of a value or they can be completely unrelated, such as $.f$ and $.g$ for instance, representing accesses to two different fields of a structure. They can also represent sequences of accesses of different depths, one being the prefix of the other, i.e. being closer to the root. For example, the path $.f$ is a prefix of the path $.f\langle i \rangle$; the first represents the access to the field f, whereas the second one represents an access to the i-th element of the array nested in the field f.

To distinguish between these cases, we define a *link type* and a *matching* operator.

Definition 7.3.3. Link Type $\mu \in \mathscr{M}$.

A link type, denoted by $\mu \in \mathscr{M}$ is defined as follows:

$$\begin{aligned} \mu := \quad & \mid \mathsf{Identical} \\ & \mid \mathsf{Left}\ \widehat{\pi} \\ & \mid \mathsf{Right}\ \widehat{\pi} \\ & \mid \mathsf{Incompatible} \end{aligned}$$

Definition 7.3.4. Matching Operator $\curlywedge$.

The matching operator $\curlywedge$ retrieves the link μ between two paths:

$$\curlywedge : \widehat{\Pi} \times \widehat{\Pi} \to \mathscr{M} \qquad \curlywedge\,(\widehat{\pi}, \widehat{\rho}) = \begin{cases} \mathsf{Identical}, & \widehat{\pi} = \widehat{\rho} \\ \mathsf{Left}\ \widehat{\pi}', & \widehat{\pi} :: \widehat{\pi}' = \widehat{\rho} \\ \mathsf{Right}\ \widehat{\rho}', & \widehat{\rho} :: \widehat{\rho}' = \widehat{\pi} \\ \mathsf{Incompatible}, & \text{otherwise} \end{cases}$$

The different cases are depicted in Table 7.11.

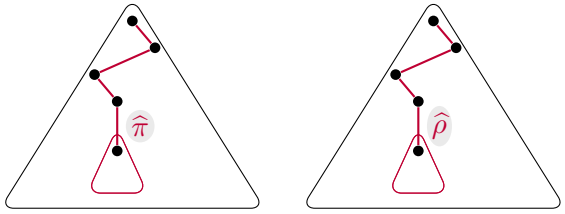
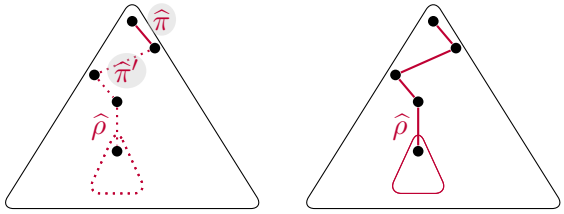
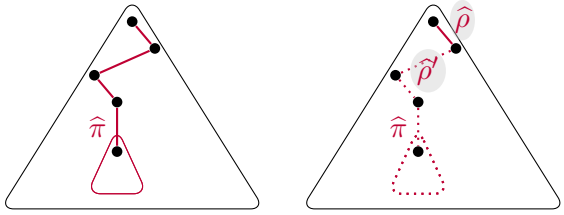
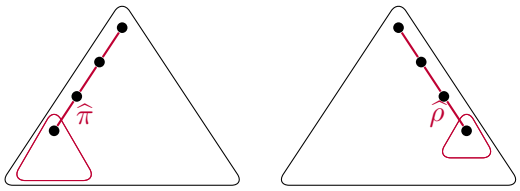
$\curlywedge(\widehat{\pi}, \widehat{\rho}) = \mathsf{Identical}$	
$\curlywedge(\widehat{\pi}, \widehat{\rho}) = \mathsf{Left}\ \widehat{\pi}'$	
$\curlywedge(\widehat{\pi}, \widehat{\rho}) = \mathsf{Right}\ \widehat{\rho}'$	
$\curlywedge(\widehat{\pi}, \widehat{\rho}) = \mathsf{Incompatible}$	

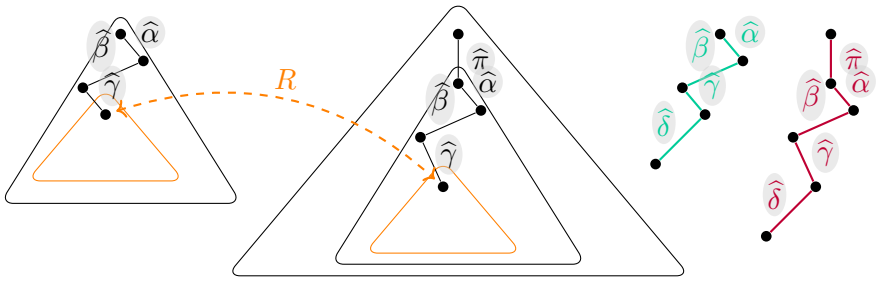
TABLE 7.11 – Links between Access Paths

Definition 7.3.5. Aligning.

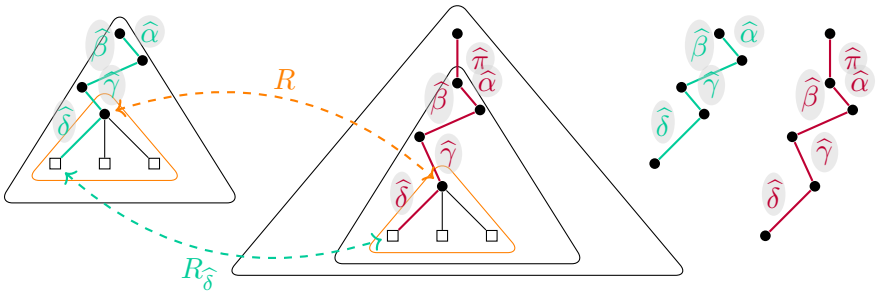
Aligning a correlation $(\widehat{\pi}, \widehat{\rho}) \mapsto R$ to another pair of paths $(\widehat{\pi}', \widehat{\rho}')$, is denoted by $\|$:

$$\| : (\widehat{\Pi} \times \widehat{\Pi} \times \mathscr{R}) \times (\widehat{\Pi} \times \widehat{\Pi}) \to \mathscr{R} \quad [(\widehat{\pi}, \widehat{\rho}) \mapsto R] \parallel (\widehat{\pi}', \widehat{\rho}') = R^{(\pi,\rho)}_{\|(\pi',\rho')}.$$

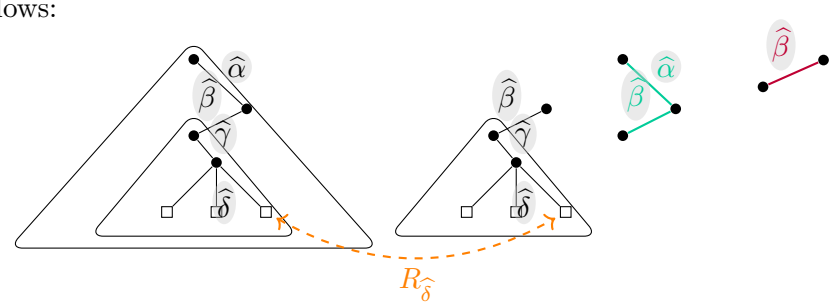
From R we obtain the information referring to the elements identified by $(\widehat{\pi}', \widehat{\rho}')$ and denote it by $R^{(\pi,\rho)}_{\|(\pi',\rho')}$. This is done by *matching* on $\widehat{\pi}$ and $\widehat{\pi}'$ on the one hand and on $\widehat{\rho}$ and $\widehat{\rho}'$ on the other and by distinguishing between the different cases. When the paths are identical, we can simply return the relation R. When the links between the paths differ or when the paths are incompatible, we have to approximate to the least precise relation, thus returning Any. When $\widehat{\pi}$ and $\widehat{\rho}$ are more shallow paths, i.e. closer to the root, we need to make a *projection*, denoted by $\rightsquigarrow$. For example, aligning $(.f, \widehat{\varepsilon}) \mapsto \{a \mapsto R_a; b \mapsto R_b; c \mapsto R_c\}$ to $(.f.b, .b)$ consists in projecting $.b$ on the relation $\{a \mapsto R_a; b \mapsto R_b; c \mapsto R_c\}$ and thus obtaining R_b. More generically, this case is depicted below:



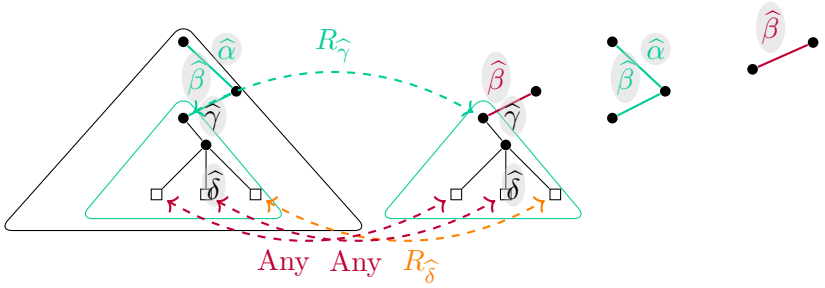
For aligning the known correlation to the given pair of paths, we need to extract from R the information that is relevant for the nested element $\widehat{\delta}$, as depicted below.



On the contrary, if $\widehat{\pi}'$ and $\widehat{\rho}'$ are closer to the root, we need to perform an *injection*, denoted by $\curvearrowleft$. For example, aligning $(.f.b, .b) \mapsto R_b$ to $(.f, \widehat{\varepsilon})$ consists in creating a relation $\{a \mapsto \mathsf{Any}; b \mapsto R_b; c \mapsto \mathsf{Any}\}$. More generically, this case can be depicted as follows:



For aligning the known correlation to the given pair of paths, we need to express the relation $R_{\widehat{\delta}}$ at the level of the $(\widehat{\alpha}\widehat{\beta}, \widehat{\beta})$ paths, a level that is closer to the root. This consists in creating a new, higher-level relation where the element identified by $\widehat{\delta}$ is mapped to $R_{\widehat{\delta}}$ and everything else is "filled" with Any since nothing is known about the rest of the elements. This can be depicted as follows:



In the general case, $R^{(\pi,\rho)}_{\|(\pi',\rho')}$ is computed as defined below.

Definition 7.3.6. Computation of $R^{(\pi,\rho)}_{\|(\pi',\rho')}$.

$$R^{(\pi,\rho)}_{\|(\pi',\rho')} = \begin{cases} R & \text{when } \curlywedge\,(\widehat{\pi},\widehat{\pi}') = \curlywedge(\widehat{\rho},\widehat{\rho}') = \mathsf{Identical} \\ \rightsquigarrow\,(\widehat{\sigma},R) & \text{when } \curlywedge\,(\widehat{\pi},\widehat{\pi}') = \curlywedge(\widehat{\rho},\widehat{\rho}') = \mathsf{Left}\ \widehat{\sigma} \\ \curvearrowleft\,(R,\widehat{\sigma}) & \text{when } \curlywedge\,(\widehat{\pi},\widehat{\pi}') = \curlywedge(\widehat{\rho},\widehat{\rho}') = \mathsf{Right}\ \widehat{\sigma} \\ \mathsf{Any} & \text{otherwise} \end{cases}$$

The used projection $\rightsquigarrow$ and injection $\curvearrowleft$ operators are defined as follows:

Definition 7.3.7. Projection Operator $\rightsquigarrow$.

$$\rightsquigarrow : \widehat{\Pi} \times \mathscr{R} \nrightarrow \mathscr{R}.$$

$$\text{Projection}: \quad \rightsquigarrow(\widehat{\pi},R) = \begin{cases} R & \text{when } \widehat{\pi} = \widehat{\varepsilon} \\ \rightsquigarrow(\widehat{\pi}', \mathit{extr}_f(R)), & \text{when } \widehat{\pi} = .f\widehat{\pi}' \\ \rightsquigarrow(\widehat{\pi}', \mathit{extr}_C(R)), & \text{when } \widehat{\pi} = @C\widehat{\pi}' \\ \rightsquigarrow(\widehat{\pi}', \mathit{extr}_{\langle i\rangle}(R)), & \text{when } \widehat{\pi} = \langle i\rangle\widehat{\pi}' \end{cases}$$

Definition 7.3.8. Injection Operator $\curvearrowleft$.

$$\curvearrowleft : \mathscr{R} \times \widehat{\Pi} \nrightarrow \mathscr{R}.$$

$$\text{Injection}: \curvearrowleft(R,\widehat{\pi}) = \begin{cases} R & \text{when } \widehat{\pi} = \widehat{\varepsilon} \\ \{f_1 \mapsto \mathsf{Any}; \ldots; f_i \mapsto \curvearrowleft(R,\widehat{\pi}'); \ldots; f_n \mapsto \mathsf{Any}\}, & \text{when } \widehat{\pi} = .f\widehat{\pi}', f = f_i \\ [C_1 \mapsto \mathsf{Any}; \ldots; C_i \mapsto \curvearrowleft(R,\widehat{\pi}'); \ldots; C_n \mapsto \mathsf{Any}], & \text{when } \widehat{\pi} = @C\widehat{\pi}', C = C_i \\ \left\langle \mathsf{Any} \triangleright i : \curvearrowleft(R,\widehat{\pi}') \right\rangle, & \text{when } \widehat{\pi} = \langle i\rangle\widehat{\pi}' \end{cases}$$

For applying the injection operator we need to know the types of the elements onto which the relation is injected, i.e. in order to "fill" the unknown relations for fields or constructors with Any, we need to know which those fields or constructors are. Thus, in practice, we need to connect the types to the context.

Aligning a correlation map $\kappa \in \mathscr{K}$ to $(\widehat{\pi}',\widehat{\rho}')$, amounts to performing this operation for each element $(\widehat{\pi},\widehat{\rho}) \mapsto R$ of κ and intersecting the results with the $\wedge_{\mathscr{R}}$ operator (Definition 7.2.4).

Definition 7.3.9. Aligning Correlation Maps.

$$\kappa \parallel (\widehat{\pi}',\widehat{\rho}') = \bigwedge_{(\widehat{\pi},\widehat{\rho})\mapsto R \in \kappa}^{\mathscr{R}} R^{(\pi,\rho)}_{\|(\pi',\rho')}.$$

The obtained results $R^{(\pi,\rho)}_{\|(\pi',\rho')}$ are intersected in order to take into account all the information scattered across the different elements of κ, and thus to obtain the most precise partial equivalence relation that is contained in κ about the elements identified by $(\widehat{\pi}', \widehat{\rho}')$.

Finally, we can define the preorder for correlation maps.

Definition 7.3.10. Correlation Maps Preorder $\hat{\sqsubseteq}$.

$$\kappa_1 \hat{\sqsubseteq} \kappa_2 \iff \forall[(\widehat{\pi}, \widehat{\rho}) \mapsto R] \in \kappa_2,\ \kappa_1 \parallel (\widehat{\pi}, \widehat{\rho}) \sqsubseteq_{\mathscr{R}} R.$$

A correlation map κ_1 is therefore more precise than another correlation map κ_2, if the relation obtained by aligning κ_1 to any pair of paths $(\widehat{\pi}, \widehat{\rho})$ of κ_2 is more precise than R, the relation mapped to this pair in κ_2. By definition, any correlation map $\kappa \in \mathscr{K}$ is smaller than $\emptyset$, the empty correlation map. Therefore, the empty correlation map is the top element for the correlation maps semilattice. A bottom element in this case does not make sense, as it would have to map to Equal any pair of paths denoting (sub)elements having compatible types.
The defined join operation between two correlation maps is denoted by $\hat{\bigvee}$.

Definition 7.3.11. Join Operation $\hat{\bigvee}$ for Correlation Maps.

$$\kappa_1 \hat{\bigvee} \kappa_2 = \kappa_3 \iff \forall[(\widehat{\pi}, \widehat{\rho}) \mapsto R] \in \kappa_1, \kappa_3(\widehat{\pi}, \widehat{\rho}) = R \vee_{\mathscr{R}} \kappa_2 \parallel (\widehat{\pi}, \widehat{\rho}).$$

It consists in aligning the correlation map κ_2 to any correlation $(\widehat{\pi}, \widehat{\rho}) \mapsto R$ in κ_1 and joining the obtained aligned relation with R. We note that the correlation map obtained by joining κ_1 and κ_2 will contain the same keys as κ_1. We could have expressed join by aligning the first correlation map to the elements of the second map. This would lead to results that have different forms, i.e. $(\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \{f \mapsto R\}$ versus $(.f, .f) \mapsto R$, but which are equivalent by definition.

The *meet* operation between two correlation maps is denoted by $\hat{\bigwedge}$.

Definition 7.3.12. Meet Operation $\hat{\bigwedge}$ for Correlation Maps.

$$\kappa_1 \hat{\bigwedge} \kappa_2 = \kappa_3 \iff \kappa_3(\widehat{\pi}, \widehat{\rho}) = \begin{cases} R \wedge_{\mathscr{R}} R', & \text{when } (\widehat{\pi}, \widehat{\rho}) \mapsto R \in \kappa_1, \\ & \text{and } (\widehat{\pi}, \widehat{\rho}) \mapsto R' \in \kappa_2 \\ R & \text{when } (\widehat{\pi}, \widehat{\rho}) \mapsto R \in \kappa_1, \\ R' & \text{when } (\widehat{\pi}, \widehat{\rho}) \mapsto R' \in \kappa_2 \end{cases} \forall(\widehat{\pi}, \widehat{\rho}).$$

7.4 Intraprocedural Correlation Analysis

7.4.1 Intraprocedural Correlation Summaries and Analysis

As was the case for the dependency analysis presented in Chapter 5, we are working with a control flow graph (CFG) representation of the predicates' bodies. We remind that nodes represent program states and edges are defined by statements with a particular exit label λ. In our case, all the outgoing edges of a node n bear the different cases of

the same statement s found at the program point n. For each statement s there is an edge labeled s, λ_k for each of its possible exit labels λ_k (as discussed in Section 4.2). However, similarly to the dependency analysis, our correlation analysis does not depend on this specificity.

Intraprocedurally, correlation information has to be kept at each point of the control flow graph, for each input and output pair of the node.

Definition 7.4.1. Intraprocedural Correlation Summaries.

An *intraprocedural* correlation summary is a mapping from pairs of variables $v \in \mathcal{V}$ to correlation maps:

$$K \in \mathcal{K}, \quad K : \mathcal{V} \times \mathcal{V} \to \mathscr{K}.$$

There is one special case, called *NoCorrelation*, which associates Any – the least precise partial relation – to any pair of variables, on any pair of valid, compatible paths. It is the top element at the intraprocedural level. *Unreachable* is used for nodes that cannot be reached, as its name implies, and constitutes the bottom element at the intraprocedural level.

For each node of a given control flow graph, $K(e, o)$ retrieves the correlation map between the local variable e and the output variable o. If a mapping for e and o does not currently exist, $K(e, o)$ retrieves the correlation $(\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}$ when $e = o$ or the empty correlation map $\emptyset$, otherwise.

Establishing the partial order $\sqsubseteq_{\mathcal{K}}$ and the join operation $\bigvee_{\mathcal{K}}$ is straightforward: $\hat{\sqsubseteq}$ (Definition 7.3.10) and $\hat{\bigvee}$ (Definition 7.3.11) are extended pointwise to an intraprocedural summary, for each ordered input-output pair and its associated correlation map.

Definition 7.4.2. Partial Order for Intraprocedural Correlation Summaries.

$$\sqsubseteq_{\mathcal{K}} \subseteq \mathcal{K} \times \mathcal{K} \qquad K_1 \sqsubseteq_{\mathcal{K}} K_2 \iff \forall e, o \in \mathcal{V},\ K_1(e, o) \mathbin{\hat{\sqsubseteq}} K_2(e, o).$$

Definition 7.4.3. Join Operation for Intraprocedural Correlation Summaries.

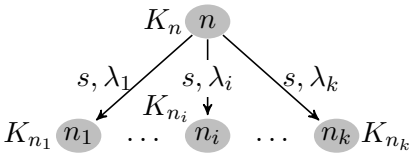
$$\bigvee_{\mathcal{K}} : \mathcal{K} \times \mathcal{K} \to \mathcal{K} \qquad K_1 \bigvee_{\mathcal{K}} K_2 = K_3 \iff \forall (e, o),\ K_3(e, o) = K_1(e, o) \mathbin{\hat{\bigvee}} K_2(e, o).$$

Our correlation analysis is a *backward* data-flow analysis, computing an intraprocedural summary at each point of the control flow graph. This represents the correlations at the node's *entry point*. For each exit label, it traverses the control flow graph starting with its corresponding exit node. The intraprocedural summary for the currently analysed label is initialized with pairs between the local value of each associated output variable of the label and the final value of the same output variable, mapped to $(\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}$. The analysis traverses the control flow graph and gradually refines the correlations, using Kildall's worklist algorithm (Kildall, 1973), until a fixed point is reached. Table 7.12 summarizes the representation and general equation of the statements. For each statement, the presented data-flow equation operates on the intraprocedural summaries of the statement's *successor* nodes. The intraprocedural summary at the *entry point* of the node is obtained by *joining* the contributions of each *outgoing* edge.

Definition 7.4.4. The contribution of an edge (n, n_i) labeled with s and λ_i is given by $\mathbb{C}^s_{\lambda_i}(K_{n_i}) \in \mathbb{C}$ where $\mathbb{C}^s_{\lambda_i}(.)$ is the *transfer function* of the edge labeled s, λ_i.

We note that there are four statements supported by αSmil, i.e. the equality test, no-operation, the partial structure equality test and the possible variant test, that have no write effects and thus have no own contribution and are not included in Table 7.12. Excepting the no-operation statement, the correlation information at their entry point is obtained by simply joining the intraprocedural summaries of their successor nodes on the true and false exit labels. For the no-operation statement, the correlation information at the entry point is identical to the intraprocedural summary of its only successor node, the one on the true exit label.

TABLE 7.12 – Statements – Representations and Data-Flow Equations

Representation	Equation
	$K_n = \bigvee_{\mathcal{K}} \mathbb{C}^s_{\lambda_i}(K_{n_i})$ $n \xrightarrow{s,\lambda_i} n_i$

Statement		$\mathbb{C}^s_\lambda(.)$: c^s_λ	$kill_\lambda$
Assignment	$o := e$	$\{(e,o) \mapsto [(\widehat{\varepsilon},\widehat{\varepsilon}) \mapsto \mathsf{Equal}]\}$	$\{o\}_{true}$
New Struct	$r := \{e_1, \ldots, e_n\}$	$\forall i, 1 \leq i \leq n \; \{(e_i, r) \mapsto [(\widehat{\varepsilon}, .f_i) \mapsto \mathsf{Equal}]\}$	$\{r\}_{true}$
Destructure	$\{o_1, \ldots, o_n\} := r$	$\forall i, 1 \leq i \leq n \; \{(r, o_i) \mapsto [(.f_i, \widehat{\varepsilon}) \mapsto \mathsf{Equal}]\}$	$\{o_i\}_{true}$
Get Field	$o := r.f_i$	$\{(r,o) \mapsto [(.f_i, \widehat{\varepsilon}) \mapsto \mathsf{Equal}]\}$	$\{o\}_{true}$
Set Field	$r' := \{r \; \textbf{\textit{with}} \; f_i = e\}$	$\{(r,r') \mapsto [(\widehat{\varepsilon},\widehat{\varepsilon}) \mapsto \{f_1 \mapsto \mathsf{Equal}; \ldots; f_i \mapsto \mathsf{Any}; \ldots; f_n \mapsto \mathsf{Equal}\}]$ $(e,r') \mapsto [(\widehat{\varepsilon}, .f_i) \mapsto \mathsf{Equal}]\}$	$\{r'\}_{true}$
Create Var.	$v := C_p[e]$	$\{(e,v) \mapsto [(\widehat{\varepsilon}, @C_p.e) \mapsto \mathsf{Equal}]\}$	$\{v\}_{true}$
Var. Switch	$\textbf{\textit{switch}}(v) \; \textbf{\textit{as}} \; [o_1 \mid \ldots \mid o_n]$	$\{(v,o_i) \mapsto [(@C_i.e, \widehat{\varepsilon}) \mapsto \mathsf{Equal}]\}$	$\{o_i\}_{\lambda_{C_i}}$
Array Get	$o := a[i]$	$\{(a,o) \mapsto [(\langle i \rangle, \widehat{\varepsilon}) \mapsto \mathsf{Equal}]\}$	$\{o\}_{true}$
Array Set	$a' := [a \; \textbf{\textit{with}} \; i = e]$	$\{(a,a') \mapsto [(\widehat{\varepsilon},\widehat{\varepsilon}) \mapsto \langle \mathsf{Equal} \triangleright i \; : \; \mathsf{Any} \rangle]$ $(e,a') \mapsto [(\widehat{\varepsilon}, \langle i \rangle) \mapsto \mathsf{Equal}]\}$	$\{a'\}_{true}$

The transfer function $\mathbb{C}^s_\lambda(.)$ formalizes the correlations created by the statement s on the label λ between its local input variables and its local output variables, denoted by c^s_λ, as well as the set $kill_\lambda$ of variables whose values have been redefined by the statement s on the label λ. These are shown in Table 7.12. There is one crucial difference between transfer functions $\mathbb{C}^s_\lambda(.)$ and intraprocedural summaries K. An intraprocedural summary K implicitly maps any pair (v,v) for $v \in \mathcal{V}$ to $(\widehat{\varepsilon},\widehat{\varepsilon}) \mapsto \mathsf{Equal}$. On the contrary, in c^s_λ, when the variable v is used as both input and output by the

statement s, the pair (v, v) is mapped to the correlation map known between the input's v old value and the output's v fresh value. Otherwise, when v is an output, i.e. $v \in \mathit{kill}_\lambda$, but not an input of s, (v, v) is mapped to $\emptyset$. We remark that K represents a *state*, while c^s_λ represents a transition.

In order to obtain the contribution $\mathbb{C}^s_{\lambda_i}(K_{n_i})$ of an edge labeled with s and λ_i, we need to connect the information given by $c^s_{\lambda_i}$ to the information contained in the intraprocedural summary K_{n_i}. For example, at the entry of node 3 in Figure 7.1 (on page 138), when considering the scenario in which the predicate exits with **true**, the intraprocedural summary contains the mapping:

$$(\texttt{th}, \texttt{o}) \;\mapsto\; \left[(\texttt{@Some.t}, \texttt{.threads}\langle \texttt{i} \rangle \texttt{@Some.t}) \mapsto \left\{ \begin{array}{rcl} \texttt{identifier} & \mapsto & \mathsf{Equal} \\ \texttt{current\_state} & \mapsto & \mathsf{Any} \\ \texttt{stack} & \mapsto & \mathsf{Equal} \end{array} \right\} \right].$$

On the **true** edge, statement 2 creates the mapping:

$$(\texttt{ta}, \texttt{th}) \;\mapsto\; [(\langle \texttt{i} \rangle, \widehat{\varepsilon}) \mapsto \mathsf{Equal}]\,.$$

Intuitively, since we are traversing the graph backwards and we are mapping *ordered* (local) input-output pairs, (ta, th) and (th, o) can be seen as a *def-use* pair: the correlation associated to (ta, th) expresses the relation between the *defined* value of th and the input ta used for creating it, while the correlation associated to (th, o) shows a subsequent *use* of that value of th for creating o. The contribution of statement 2 on the **true** edge should capture this flow of ta's value to o's value, through the variable th. Thus, it should contain a mapping for the pair (ta, o). In the general case we need to detect any variable r such that $[(p, r) \mapsto \kappa] \in c^s_{\lambda_i}$, $[(r, q) \mapsto \kappa'] \in K_{n_i}$ and compute the mapping for (p, q) in $\mathbb{C}^s_{\lambda_i}(K_{n_i})$.

In order to compute the correlation map associated to (ta, o), we take into account the fact that both the right path $\widehat{\varepsilon}$ of $c^s_\lambda(\texttt{ta}, \texttt{th})$ and the left path @Some.t of K_{n_3}(th, o) refer to the th variable. However, they do not represent traversals of the same depth: $\widehat{\varepsilon}$ refers to the entire value of th, while @Some.t refers to the value below the constructor Some. Between ta and o we can conclude that the values nested under the Some constructor of the i-th elements are related:

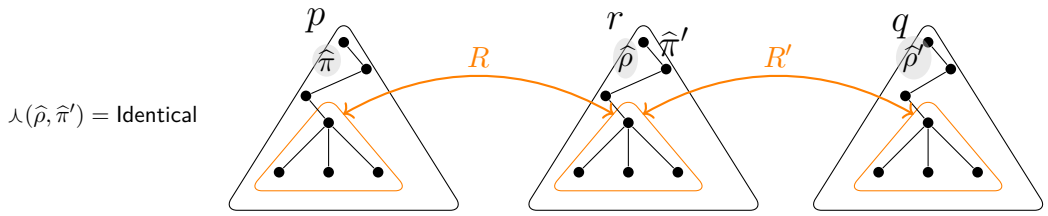
$$(\texttt{ta}, \texttt{o}) \;\mapsto\; \left[\langle \texttt{i} \rangle \texttt{@Some.t}, \texttt{.threads}\langle \texttt{i} \rangle \texttt{@Some.t}) \mapsto \left\{ \begin{array}{rcl} \texttt{identifier} & \mapsto & \mathsf{Equal} \\ \texttt{current\_state} & \mapsto & \mathsf{Any} \\ \texttt{stack} & \mapsto & \mathsf{Equal} \end{array} \right\} \right].$$

We call the process of obtaining the correlation map associated to (ta, o) from the correlations associated to (ta, th) and (th, o) *composition*.

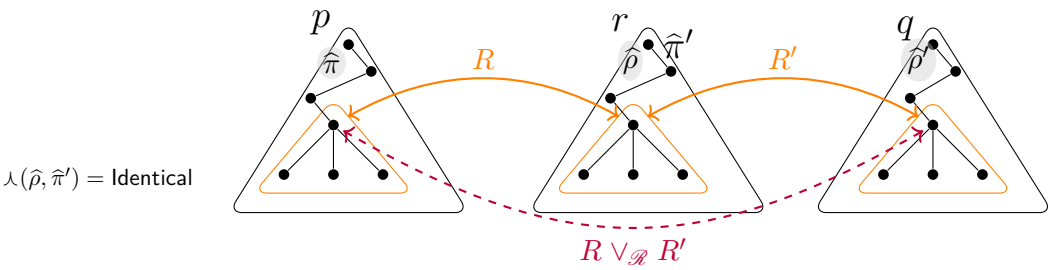
In the general case, the composition operation is denoted by $\odot$ and it refers to the process of computing the flow of a variable p to a variable q through an intermediate variable r. Thus, when knowing that $(p, r) \mapsto [(\widehat{\pi}, \widehat{\rho}) \mapsto R]$ and that $(r, q) \mapsto [(\widehat{\pi}', \widehat{\rho}') \mapsto R']$, we must first obtain the *link* (Definition 7.3.3) between the paths $\widehat{\rho}$ and $\widehat{\pi}'$ relating

subvalues of r to subvalues of p and q, respectively. This is obtained by *matching* with $\curlywedge$ (Definition 7.3.4). In the context of the example given above, $\widehat{\rho}$ and $\widehat{\pi}'$ are the paths referring to subvalues of the `th` variable, i.e. $\widehat{\varepsilon}$ and `@Some.t`, respectively. If the two paths are incompatible, i.e. they refer to different, unrelated subvalues of r, there is no flow between p and q through r. If the paths are compatible, we can compute the correlation between p and r, by distinguishing between the three different possible link cases obtained with $\curlywedge$.

The case when the same subvalue of r identified by $\widehat{\rho}$ (and the identical $\widehat{\pi}'$) is related to both p and q is depicted below:

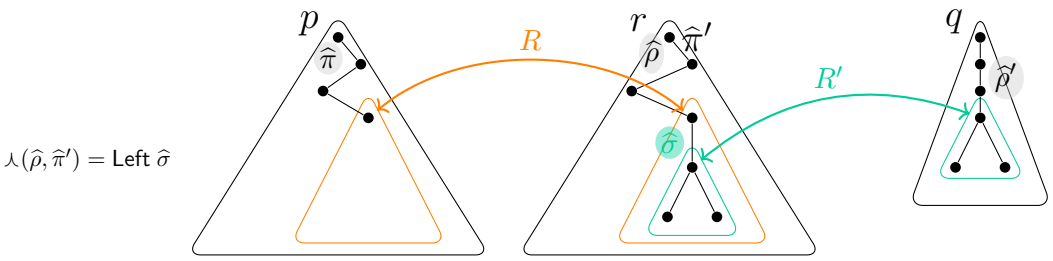


In this case, computing the flow from p to q through r is rather straightforward. Since the same subvalue of r is related to p's subvalue identified by $\widehat{\pi}$ and to q's subvalue identified by $\widehat{\rho}'$, we can relate these two subvalues and map the pair $(\widehat{\pi}, \widehat{\rho}')$ to the relation obtained by composing R and R'. We note that given the special form of partial relations $R \in \mathscr{R}$, the compose operation at this level is equivalent to $\vee_{\mathscr{R}}$[1] (Definition 7.2.3). The computation of the correlation for p and q is depicted below:

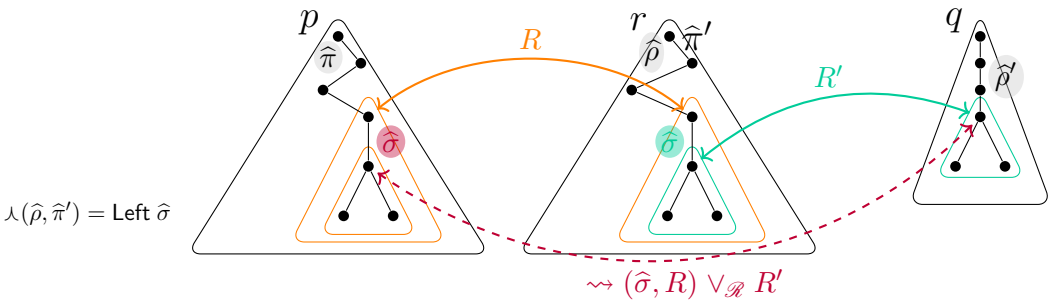


The subelements of r related to p and to q respectively can also have different granularities, one being nested deeper in r than the other. For instance, the subvalue of r identified by the path $\widehat{\rho}$ can be closer to the root than its subelement identified by $\widehat{\pi}'$ related to q. This case is depicted below:

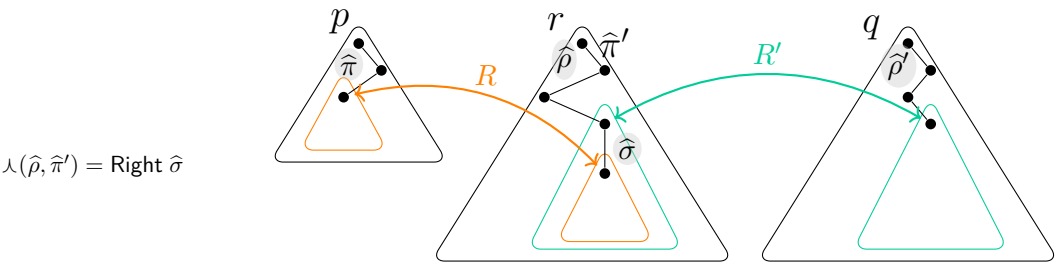
[1] However, this would not be the case anymore for a more complex partial relation type, including not only equivalences but also more general relations.



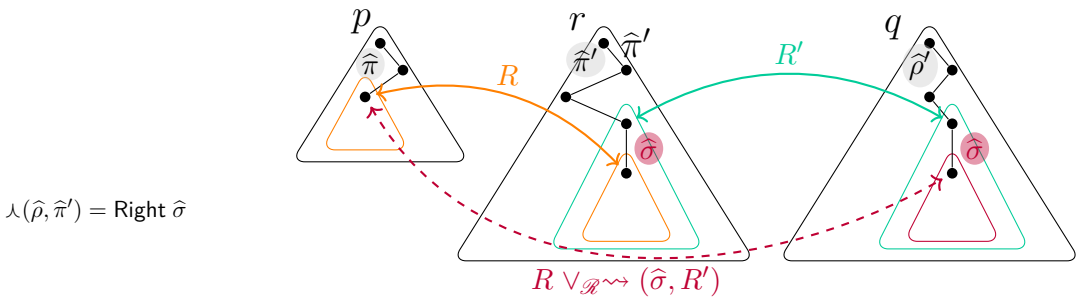
In this case we can only detect the flow of p to q at the level of r's subelement that is related to both p and q, i.e. the subelement nested deeper. Thus, in order to compute the correlation between p and q, we need to project $\widehat{\sigma}$ on R, and to compose the obtained relation with R'. This is summarized by the following figure:



Finally, in the complementary case, the subvalue of r identified by the path $\widehat{\rho}$ and correlated to p can be nested deeper than the subvalue identified by $\widehat{\pi}'$ which is correlated to q. This case is depicted below:



As in the previous case, we can only detect the flow of p to q at the level of r's subelement that is related to both p and q, i.e. the subelement nested deeper. In this case we need to project $\widehat{\sigma}$ on R' and to compose the obtained relation with R. The flow between p and q is at the level of the subvalues identified by $\widehat{\pi}$ and $\widehat{\rho}' :: \widehat{\sigma}$ respectively. This is illustrated below:



Formally, if the $\widehat{\rho}$ and $\widehat{\pi}'$ paths are compatible, we compose the correlation elements $(\pi, \rho) \mapsto R$ and $(\pi', \rho') \mapsto R'$, thereby obtaining a new correlation element, $(\pi_\bullet, \rho_\bullet) \mapsto R_{\bowtie}$, which is computed as shown below.

Definition 7.4.5. Computing $(\pi_\bullet, \rho_\bullet) \mapsto R_{\bowtie}$.

$$(\pi_\bullet, \rho_\bullet) = (\pi, \rho) \bullet (\pi', \rho') \stackrel{def}{=} \begin{cases} (\pi, \rho') & \text{when } \curlywedge\,(\rho, \pi') = \mathsf{Identical} \\ (\pi :: \sigma, \rho') & \text{when } \curlywedge\,(\rho, \pi') = \mathsf{Left}\ \sigma \\ (\pi, \rho' :: \sigma) & \text{when } \curlywedge\,(\rho, \pi') = \mathsf{Right}\ \sigma \end{cases}$$

$$R_{\bowtie} = R \bowtie R' \stackrel{def}{=} \begin{cases} R \vee_{\mathscr{R}} R' & \text{when } \curlywedge\,(\rho, \pi') = \mathsf{Identical} \\ \rightsquigarrow (\sigma, R) \vee_{\mathscr{R}} R' & \text{when } \curlywedge\,(\rho, \pi') = \mathsf{Left}\ \sigma \\ R \vee_{\mathscr{R}} \rightsquigarrow (\sigma, R') & \text{when } \curlywedge\,(\rho, \pi') = \mathsf{Right}\ \sigma \end{cases}$$

We note that the use of the projection operation $\rightsquigarrow$ (Definition 7.3.7) for both compatible, non-identical link cases for r's access paths related to p and to q respectively, is a consequence of not choosing a canonical form for correlations. The flexibility confered by the absence of a canonical correlation form is visible at the composition level.

The composition of correlation maps is denoted by $\bigcirc$ and defined below.

Definition 7.4.6. Composition of Correlation Maps.

Computing $\kappa_1 \bigcirc \kappa_2$ amounts to intersecting the composition of all correlation elements from κ_1 and κ_2:

$$(\kappa_1 \bigcirc \kappa_2)(\pi_\bullet, \rho_\bullet) = \bigwedge_{\substack{\mathscr{R} \\ (\pi,\rho)\mapsto R \in \kappa_1 \\ (\pi',\rho')\mapsto R' \in \kappa_2 \\ (\pi_\bullet,\rho_\bullet)=(\pi,\rho)\bullet(\pi',\rho')}} R \bowtie R'.$$

Finally, the contribution $\mathbb{C}^s_{\lambda_i}(K_{n_i})$ is obtained as defined below.

Definition 7.4.7. Contribution $\mathbb{C}^s_{\lambda_i}(K_{n_i})$.

$$\odot : \mathbb{C} \times \mathcal{K} \to \mathcal{K} \quad c^s_\lambda \odot K = K' \quad \text{where } K'(p, q) = \hat{\bigwedge_r}(c^s_\lambda(p, r) \bigcirc K(r, q)).$$

It is depicted in Figure 7.5.

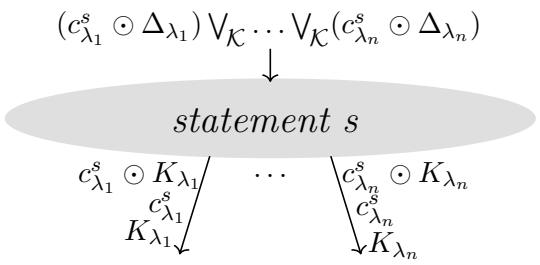


FIGURE 7.5 – Entry Point – Correlation Information

We conclude this section by specifying what it means for intraprocedural correlation summaries to be *well-formed*, showing the corresponding inference rule in Table 7.19. Only ordered input-output pairs can appear as keys in intraprocedural mappings. Therefore, the well-formedness judgement is parameterized by the set of input variables $\mathcal{I}$, and by the set of output variables $\mathcal{O}$. The former indicate variables that have the right to appear as left members of the variable pairs, while the latter indicate variables that have the right to appear as right members of the variable pairs. The correlation map associated to each such input-output pair must be well-typed with respect to the types of the variables as given by the typing environment Γ (Definition 4.3.1). The typing judgement for correlation maps was shown in Table 7.9.

$$\frac{\forall (e,o) \mapsto \kappa \in K \qquad \Gamma(e) = \tau_e \qquad \Gamma(o) = \tau_o \qquad e \in \mathcal{I} \qquad o \in \mathcal{O} \qquad \Gamma, \mathcal{I} \vdash \kappa \ : (\tau_e, \tau_o)}{\Gamma, \mathcal{I}, \mathcal{O} \vDash K} \ \text{WFIntraCor}$$

TABLE 7.19 – Well-Formed Intraprocedural Correlation Summaries

7.4.2 Intraprocedural Correlation Analysis Illustrated

To better illustrate our correlation analysis at an intraprocedural level and to summarize everything that has been presented so far in this chapter, we exemplify the mechanism behind it, step by step, on the predicate `stop_thread`, discussed in Section 7.1.1 on page 138. We consider the **true** execution scenario, apply our analysis and compare the actual obtained correlation results with the targeted ones depicted in Figure 7.2.

Since a predicate can only exit with one label at a time and we are analysing the **true** label, we can map the exit node `inval` to the special case *Unreachable*. We begin by initialising the correlation summary for the exit node corresponding to the **true** exit label. As shown in Figure 7.6, this consists in mapping the pair referring to the *local* value of the `o` variable and the *final* state of `o`, to a correlation map containing a single correlation, namely $(\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}$. This acknowledges that the value of the output `o` retrieved to the predicate's callers is the most recent value computed locally. In the following, we denote the final value of `o` by `o#` in order to distinguish it from the local value.

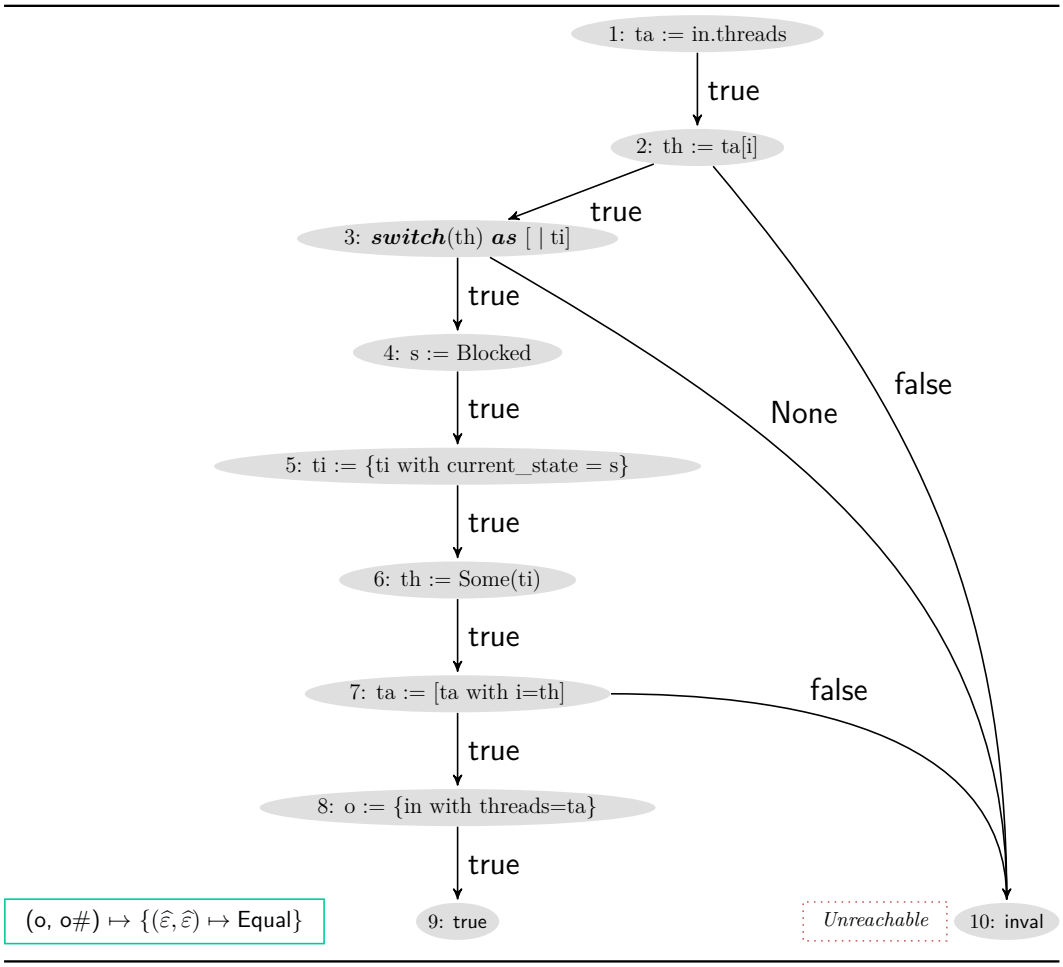


FIGURE 7.6 – Analysing Predicate `stop_thread` – Initialisation

We advance backwards along the control flow graph, reaching node 8. We apply the equation corresponding to a field access as given in Table 7.12 and obtain the following correlation summary:

$$(\mathsf{in},\mathsf{o}) \;\mapsto\; \left\{ (\widehat{\varepsilon},\widehat{\varepsilon}) \mapsto \left\{ \begin{array}{rcl} \mathsf{threads} & \mapsto & \mathsf{Any} \\ \mathsf{pid} & \mapsto & \mathsf{Equal} \\ \mathsf{crt\_thread} & \mapsto & \mathsf{Equal} \\ \mathsf{adr\_space} & \mapsto & \mathsf{Equal} \end{array} \right\} \right\} .$$

$$(\mathsf{ta},\mathsf{o}) \;\mapsto\; \{(\widehat{\varepsilon},.\mathsf{threads}) \mapsto \mathsf{Equal}\}$$

We compose it with the correlation summary of its successor node, i.e. the exit node corresponding to the **true** exit label, thus detecting the flow of `in` to `o#` and of `ta` to `o#`

respectively, through the local value `o`. This amounts to:

$$(\mathsf{in}, \mathsf{o}\#) \quad \mapsto \quad \left\{ (\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \left\{ \begin{array}{rcl} \mathsf{threads} & \mapsto & \mathsf{Any} \\ \mathsf{pid} & \mapsto & \mathsf{Equal} \\ \mathsf{crt\_thread} & \mapsto & \mathsf{Equal} \\ \mathsf{adr\_space} & \mapsto & \mathsf{Equal} \end{array} \right\} \right\}.$$

$$(\mathsf{ta}, \mathsf{o}\#) \quad \mapsto \quad \{(\widehat{\varepsilon}, .\mathsf{threads}) \mapsto \mathsf{Equal}\}$$

Since node 8 does not have any other successor nodes, the correlation information at its entry point is identical to the one we have just computed.

We advance one step, reaching node 7 and apply the corresponding equation, thereby obtaining:

$$\begin{array}{rcl} (\mathsf{ta}, \mathsf{ta}) & \mapsto & \{(\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \langle \mathsf{Equal} \triangleright i : \ \mathsf{Any} \rangle\} \\ (\mathsf{th}, \mathsf{ta}) & \mapsto & \{(\widehat{\varepsilon}, \langle i \rangle) \mapsto \mathsf{Equal}\} \end{array}.$$

We compose it with the correlation summary of node 8, tracking the flow of the *local* value of `ta` to `o#`, through the *new* state of the variable `ta`, after updating its `i`-th element. We also track the flow of `th` to `o#`. The correlation map for the $(\mathsf{in}, \mathsf{o}\#)$ pair remains unchanged. We thus obtain:

$$(\mathsf{in}, \mathsf{o}\#) \quad \mapsto \quad \left\{ (\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \left\{ \begin{array}{rcl} \mathsf{threads} & \mapsto & \mathsf{Any} \\ \mathsf{pid} & \mapsto & \mathsf{Equal} \\ \mathsf{crt\_thread} & \mapsto & \mathsf{Equal} \\ \mathsf{adr\_space} & \mapsto & \mathsf{Equal} \end{array} \right\} \right\}.$$

$$(\mathsf{ta}, \mathsf{o}\#) \quad \mapsto \quad \{(\widehat{\varepsilon}, .\mathsf{threads}) \mapsto \langle \mathsf{Equal} \triangleright i : \ \mathsf{Any} \rangle$$

$$(\mathsf{th}, \mathsf{o}\#) \quad \mapsto \quad \{(\widehat{\varepsilon}, .\mathsf{threads}\langle i \rangle) \mapsto \mathsf{Equal}\}$$

In order to obtain the correlation information at the entry point of node 7, we need to join the computed correlation summary with the correlation summary known for the other successor of node 7, namely the exit node 10. Since the latter is *Unreachable*, the identity element for join at the intraprocedural level, it does not affect the correlation summary at the entry point of node 7. We proceed similarly for nodes 6, 5, 4, 3 and 2, applying the corresponding data-flow equation for each statement and composing with the intraprocedural correlation summary of the successor node. Since each of these nodes has only one possible exit label there are not multiple contributions that need to be joined. At the entry point of node 6, for example we obtain the following summary:

$$(\mathsf{ta}, \mathsf{o}\#) \quad \mapsto \quad \{(\widehat{\varepsilon}, .\mathsf{threads}) \mapsto \langle \mathsf{Equal} \triangleright i : \ \mathsf{Any} \rangle$$

$$(\mathsf{ti}, \mathsf{o}\#) \quad \mapsto \quad \{(\widehat{\varepsilon}, .\mathsf{threads}\langle i \rangle @\mathsf{Some.t}) \mapsto \mathsf{Equal}\}$$

$$(\mathsf{in}, \mathsf{o}\#) \mapsto \left\{ (\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \left\{ \begin{array}{rcl} \mathsf{threads} & \mapsto & \mathsf{Any} \\ \mathsf{pid} & \mapsto & \mathsf{Equal} \\ \mathsf{crt\_thread} & \mapsto & \mathsf{Equal} \\ \mathsf{adr\_space} & \mapsto & \mathsf{Equal} \end{array} \right\} \right\} .$$

We skip some steps and obtain the following correlation summary at the entry point of node 2:

$$(\mathsf{in}, \mathsf{o}\#) \mapsto \left\{ (\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \left\{ \begin{array}{rcl} \mathsf{threads} & \mapsto & \mathsf{Any} \\ \mathsf{pid} & \mapsto & \mathsf{Equal} \\ \mathsf{crt\_thread} & \mapsto & \mathsf{Equal} \\ \mathsf{adr\_space} & \mapsto & \mathsf{Equal} \end{array} \right\} \right\}$$

$$(\mathsf{ta}, \mathsf{o}\#) \mapsto \left\{ \begin{array}{rcl} (\widehat{\varepsilon}, .\mathsf{threads}) & \mapsto & \langle \mathsf{Equal} \triangleright i : \mathsf{Any} \rangle \\ (\langle i \rangle @\mathsf{Some.t}, .\mathsf{threads}\langle i \rangle @\mathsf{Some.t}) & \mapsto & \left\{ \begin{array}{rcl} \mathsf{id} & \mapsto & \mathsf{Equal} \\ \mathsf{current\_state} & \mapsto & \mathsf{Any} \\ \mathsf{stack} & \mapsto & \mathsf{Equal} \end{array} \right\} \end{array} \right\} .$$

Finally, we reach node 1, where we apply the data-flow equation corresponding to a field access and compose the obtained information with the correlation summary computed at the entry of node 2. We obtain:

$$(\mathsf{in}, \mathsf{o}\#) \mapsto \left\{ \begin{array}{rcl} (\widehat{\varepsilon}, \widehat{\varepsilon}) & \mapsto & \left\{ \begin{array}{rcl} \mathsf{threads} & \mapsto & \mathsf{Any} \\ \mathsf{pid} & \mapsto & \mathsf{Equal} \\ \mathsf{crt\_thread} & \mapsto & \mathsf{Equal} \\ \mathsf{adr\_space} & \mapsto & \mathsf{Equal} \end{array} \right\} \\ (.\mathsf{threads}, .\mathsf{threads}) & \mapsto & \langle \mathsf{Equal} \triangleright i : \mathsf{Any} \rangle \\ (.\mathsf{threads}\langle i \rangle @\mathsf{Some.t}, .\mathsf{threads}\langle i \rangle @\mathsf{Some.t}) & \mapsto & \left\{ \begin{array}{rcl} \mathsf{id} & \mapsto & \mathsf{Equal} \\ \mathsf{current\_state} & \mapsto & \mathsf{Any} \\ \mathsf{stack} & \mapsto & \mathsf{Equal} \end{array} \right\} \end{array} \right\}$$

Since the node 1 has only one successor node, this correlation summary represents the correlation information at the entry point of node 1, i.e. there is no other correlation summary to join it with. This contains a single pair of variables, (in, o#) and their associated correlation map. Since the pair is an input-output pair of the `stop_thread` predicate, we do not need to filter anything out. This constitutes the final correlation summary for the analysed predicate on the **true** exit label. These results are identical to the ones we had depicted as our targeted results in Figure 7.2.

For the `inval` exit label, the corresponding correlation summary is *NoCorrelation*. This example can be tried on the web page[2] dedicated to our correlation analysis. Other

[2]Correlation Analysis Web Page: http://www.ajl-demo.fr/2016/

examples are provided and explained there as well. Additionally, users can devise and test their own examples.

7.5 Interprocedural Correlation Analysis

Our analysis is performed label by label and interprocedural correlation domains associate an intraprocedural summary to each exit label of the analysed predicate. Therefore, interprocedural domains encapsulate an intraprocedural summary for each possible execution scenario of a predicate.

An interprocedural domain $\mathcal{K}_p$ of a predicate p is thus defined as shown below.

Definition 7.5.1. Interprocedural Correlation Domain.

$$\mathcal{K}_p : \Lambda_p \to \mathcal{K} \quad \text{where } \Lambda_p \text{ is the set of output labels of predicate } p.$$

The intraprocedural summary associated to each label is *filtered* so as to contain only ordered pairs of variables where the left member is an input of the analysed predicate and the right member is an output associated to the analysed label. The correlation maps associated to such pairs are built so as to contain correlations where only input variables may appear in array cell paths. Similarly, the exception index in partial equivalence relations of arrays must be an input variable. Registering exceptions in array correlations only for input variables is not a consequence of a language restriction on array operations, but simply a consequence of the fact that at the interprocedural level, only correlation information between inputs and outputs makes sense.

The interprocedural domain of a predicate is used for deducing the transfer functions for a predicate call statement.

In the following we detail the equation corresponding to a call to a predicate:

$$\underbrace{p(e_1, \ldots, e_n)[\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]}_{s}$$

having the following signature:

$$p(\epsilon_1, \ldots, \epsilon_n)[\lambda_1 : \bar{\omega}_1 \mid \ldots \mid \lambda_m : \bar{\omega}_m].$$

The general equation form given in Table 7.12 applies:

$$K_n = \bigvee_{\mathcal{K}} \limits_{n \xrightarrow{s,\lambda_i} n_i} \mathbb{C}^s_{\lambda_i}(K_{n_i}).$$

The transfer functions for the predicate call statement are deduced from the predicate's interprocedural domain in the following fashion:

$$\mathbb{C}^s_{\lambda_i}(K_{n_i}) = c^s_{\lambda_i} \odot K_{n_i},\ kill_{\lambda_i} = \{\bar{o}_i\}$$
$$c^s_{\lambda_i}(e_j, o^k_i) = \kappa^{j,k}_i, \forall j \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, h\}$$

where

$$\kappa_i^{j,k} = \mathcal{K}_p(\lambda_i)(\epsilon_j, \omega_i^k) \blacktriangleleft (\bar{\epsilon} \mapsto \bar{e})$$
$$s = p(e_1, \ldots, e_n)\ [\lambda_1 : \bar{o}_1 \mid \ldots \mid \lambda_m : \bar{o}_m]; \quad \bar{o}_i = \{o_i^1, \ldots, o_i^h\}.$$

Namely, the contribution of a predicate call to each (e_j, o_i^k) input-output pair stems from the contribution of the interprocedural domain for label λ_i and formal input-output pair (ϵ_j, ω_i^k). In these, all the formal input parameters $\bar{\epsilon}$ in array partial equivalences and in array cell paths are substituted by the corresponding effective input parameters from $\bar{e}$ or approximated away. The substitution operation is denoted by $\blacktriangleleft(\chi)$ where χ is a substitution from formal to effective parameters.

Our correlation analysis is context-insensitive and αSmil programs are analysed by computing, once and for all an interprocedural correlation summary for every predicate they contain. The correlation summaries are stored in a mapping binding predicate identifiers to their interprocedural correlation information.

7.6 Extension – Constructor Evolution

The correlation analysis as presented so far in this chapter tracks and detects partial equivalence relations between inputs and outputs of predicates. An interesting direction to investigate, would be an extension of our analysis allowing us to detect not only equivalences but more general relations, that could capture the *evolution* of constructors for variants. In Figure 7.4-b), we illustrated the form of correlations computed for variants. With the extension, the correlation information obtained for variants would be richer, as illustrated in Figure 7.7.

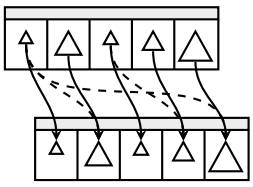


FIGURE 7.7 – Construction Evolution

This extension would allow inferring the preservation of certain properties when transitioning from a "stronger" state to a "weaker" state. For instance, we consider again our `process` and `thread` data types introduced in Chapter 3, Section 3.1.5 (on page 49 and 48, respectively). Additionally, we consider a predicate `kill_thread`, shown below, which modifies the array of associated threads of the input `p` by setting the `i`-th element to `None`. If the `i`-th element is already inactive, no modifications are made. In this case, the predicate exits with label `inactive` and simply copies `p` to the output `o`.

```
predicate kill_thread (process p, int i)
-> [true: process o | inactive: process o | oob]
{{array<option<thread>> threads, option<thread> thi, thread ti}}
{
o := p : [true -> 1];
```

```
threads := o.threads : [true -> 2];
thi := threads[i] : [true -> 3, false -> 9];
switch(thi) as [ti |] : [Some -> 4, None -> 8];
thi := None : [true -> 5];
threads := [threads with i = thi] : [true -> 6, false -> 9];
o := {o with threads = threads} : [true -> 7];
[true];
[inactive];
[oob]
}
```

For variants we are currently detecting equivalence relations between the arguments of variant values built with the same constructor. With the extension for capturing constructor evolution, we could take a step further and also detect, for a given execution scenario, the set of possible transitions between the different constructors. For instance, for the `kill_thread` predicate on the **true** exit label, we could detect that the only possible transition of the `i`-th element of the `threads` array is from `Some` to `None`. Had the element been `None`, the predicate would have followed the `inactive` execution scenario.

We further consider a predicate `disjoint_stacks(process p)`, verifying a fundamental property of any process, namely the fact that the stacks of all associated threads of the process are disjoint. If the property holds for the input process `p` prior to executing `kill_thread`, intuitively it should continue to hold subsequently, for the output process `o` as well. If the array's `i`-th element was already inactive, i.e. `None`, the property `disjoint_stack` obviously still holds since the input `p` is simply copied to the output `o`. If it was active, the transition from `Some` to `None` does not impact the property, as it does not create a new memory region that could threaten the property. In this case, the transition from `Some` to `None` is a transition from a "stronger" state to a "weaker" state.

We have conducted preliminary experiments targeting the detection of such information and these have led to promising results. Tracking general relations that capture evolution requires certain modifications that are confined to the abstract partial relation type and to the data-flow equations concerning variants.

The abstract partial relation type presented in Section 7.2 (Definition 7.2.1) would need to be extended with Impossible, an additional atomic case along with Equal and Any. It is required for signalling impossible transitions between variant constructors and leads to some overlap with the *possible-constructors* analysis presented in Chapter 5. The partial relations for variants would be expressed as a square matrix of constructors, where each element a_{C_i,C_j} of the matrix has a corresponding associated partial relation R_{C_i,C_j}. Impossible would be associated to any element a_{C_i,C_j} for which the transition from C_i to C_j is impossible. For the elements a_{C_i,C_i} on the main diagonal for which the transition from C_i to C_i is possible we could compute partial equivalences between the arguments of the C_i constructor. For the elements a_{C_i,C_j} lying outside the main matrix diagonal, for which the transition from C_i to C_j is possible the associated relation would be Any. Alternatively, for computing reflexive relations, we could consider that transitions on the main diagonal, i.e. from C_i to C_i, are always possible.

Impossible would become the bottom element of our partial relation type R, replacing Equal in this role. It would also become the identity element for the join operation $\vee_{\mathscr{R}}$ (Definition 7.2.3) of partial relations and the absorbing element for the meet operation $\wedge_{\mathscr{R}}$ (Definition 7.2.4). Similarly to the case of $\oslash$ for the abstract dependency type, the current bottom element Equal would become the middle element of a double diamond-shaped abstract type and it would require the addition of some extra comparison cases for $\sqsubseteq_{\mathscr{R}}$ (Definition 7.2.2), as well as some extra cases for the $\vee_{\mathscr{R}}$ (Table 7.2) and $\wedge_{\mathscr{R}}$ (Table 7.3) operations. The most important modification however would be in the case of the compose operation. Currently, the compose operation at the level of partial equivalence relations is $\vee_{\mathscr{R}}$. With this extension it would amount to a matrix multiplication.

7.7 Related Work

A rigorous presentation of the frame problem in specification and the different existing approaches for addressing it has been given by Borgida et al. (Borgida, Mylopoulos, and Reiter, 1993; Borgida, Mylopoulos, and Reiter, 1995). A more recent overview of framing is included in (Hatcliff et al., 2012).

In recent years, a vast body of research has been conducted on the specification of frame properties in the context of modular programming. This ranges from complex approaches imposing the *swinging pivots requirement* (Leino and Nelson, 2002), to approaches using *data groups* (Leino, 1998; Leino, Poetzsch-Heffter, and Zhou, 2002), adopting the *Universe* type system (Müller, 2002; Müller, Poetzsch-Heffter, and Leavens, 2003) or variations of it (Leino and Müller, 2004; Leino and Müller, 2006; Barnett and Naumann, 2004; Barnett et al., 2004), to approaches based on the *dynamic frame theory* (Kassios, 2006; Kassios, 2011; Smans, Jacobs, and Piessens, 2012), regional logic (Banerjee, Naumann, and Rosenberg, 2008) or separation logic (Reynolds, 2002; O'Hearn, Yang, and Reynolds, 2004; Parkinson and Bierman, 2005).

In (Smans, Jacobs, and Piessens, 2012) Smans et al. present a technique for frame inference based on a variant of dynamic frames inspired by separation logic, and relying on accessibility information contained within pre- and postconditions. By including accessibility information in a method's precondition, an upper bound on the set of locations modifiable by the method can be detected. In our case, the upper bound on the set of elements that a predicate may modify when exiting with a particular exit label is implicitly the set of output variables generated on that exit label, joined with the set of local variables. The implicit dynamic frame approach requires the specification of accessibility information. Our correlation analysis is entirely automatic and infers fine-grained frame properties for compound data structures.

The literature on shape analysis (Calcagno et al., 2009; Sagiv, Reps, and Wilhelm, 1999; Jones and Muchnick, 1979; Montenegro, Peña, and Segura, 2015) and side effects analyses (Salcianu and Rinard, 2005; Milanova, Rountev, and Ryder, 2005) is vast. The former is aimed at deep-heap mutations, while we are focusing on deep-state modifications, in the context of complex transition systems. The latter determine memory

locations that may be modified by an operation. Reasoning about heap locations is beyond our scope. We treat mappings between variables and their values, analyse their evolution in a side-effect free environment and detect not only what is modified, but also how and to what extent.

In (Chang and Leino, 2005), Chang and Leino present the congruence-closure abstract domain, designed for an object-oriented context and implemented in the Spec# program verifier. They infer and express relations between fields of variables, a goal similar to ours. The congruence-closure domain maintains equivalence graphs mapping field accesses to symbolic locations. On its own, this domain allows the inference and expression of relations for accessed fields. In order to take into account updates as well, this needs to use the heap succession domain as a base. Unlike us, they can express preorders between fields, depending on the base domains used. However, our domain handles both accesses and updates to structures, arrays and variants in a uniform manner, independent of additional information. We have sketched an extension for handling not only equivalences but also more general relations capturing constructor evolution. This is a direction we plan to investigate in the future.

Rakamarić and Hu report in (Rakamaric and Hu, 2008) a method to infer frame axioms of procedures and loops based on static analysis. As a starting point, they use the DSA shape analysis, presented by Lattner et al. (Lattner, Lenharth, and Adve, 2007). DSA provides a summary of points-to relations as a graph, that is used to compute a set of memory locations that are modified by a procedure or its callees. By a pass through the graph, for each node reachable from the globals or procedure parameters, they generate expressions representing a path to that node. The generated frame axioms are used internally by an extended static checker of C programs, i.e. in a purely automatic setting. In contrast, our analysis is designed for an interactive verification context. Our technique focusing on a purely functional language is not concerned by aliasing and does not depend on an external points-to framework.

In (Taghdiri, Seater, and Jackson, 2006), Taghdiri et al. present a technique for extracting procedure summaries for object-oriented procedures, used to prove verification conditions. Procedures are executed symbolically and the environment of the post-state is computed so as to express every variable and field in terms of the values of the variables and fields of the pre-state. Their goal is broader than ours. However, unlike their summaries, our correlation results encompass only information that is visible from the outside (to the callers).

Bertrand Meyer presents the *double frame inference* strategy, an approach that targets the automation of both frame specification and frame verification in the context of Eiffel (Meyer, 1991). The first component – the frame specification inference – relies on the analysis of method postconditions. The idea stems from an informal review of JML code, which showed that in practice there is a considerable overlap between what is mentioned in an *assignable* clause, i.e. modifies clause, and what is included in the postcondition. It relies on the observation that in general, when manually written specifications include clauses about what changes, they also include clauses about how it changes. By analysing a method's p postcondition, a set $\bar{p}$ is obtained. This represents an overapproximation of the set of elements that are allowed to be modified

by p according to its specification. The second component of the strategy, *the frame implementation inference* relies on the frame calculus (Kogtenkov, Meyer, and Velder, 2015), which is itself based on alias calculus (Kogtenkov, Meyer, and Velder, 2015; Meyer, 2010; Meyer, 2011). Methods are analysed and $\underline{p}$ is detected; this represents an overapproximation of the set of expressions whose values may change as a result of executing p. Frame verification amounts to verifying that $\bar{p}$ includes $\underline{p}$. Though our goal is closely related to the issue addressed by the double frame inference in general, and the frame calculus in particular, the approaches are not directly comparable as they target languages with different characteristics, which in turn influence both the adopted analysis techniques and the derivative targeted issues. Both approaches are conservative and automatic, i.e. neither requires manual annotations. In contrast to the frame calculus, our correlation analysis is standalone and it is not concerned by aliasing.

7.8 Conclusion

Identifying precise information concerning the effects of program operations is possible by means of static analysis without sacrificing scalability. In this chapter, we have presented a data-flow analysis that tracks the origin of subparts of the output and relates it to subparts of the inputs, thus detecting not only what is modified, but also how it is modified and to what extent. The correlation analysis is a flow-sensitive, path-sensitive, interprocedural analysis that handles arrays, structures and variants. The analysis is context-insensitive but this trait does not have a costly impact in terms of precision. We have defined a partial equivalence type mirroring the layered structure of algebraic data types and associative arrays and we introduced an intermediate level consisting of access paths and correlations in order to compute expressive, fine-grained equivalences between parts of the inputs and parts of the outputs in a flexible manner. Just as frame properties specified by means of *old* expressions tend to lead to a proliferation of conditions to be specified, our correlation summaries showing equivalences between input and output subelements can become verbose in the case of predicates handling large compound values and modifying only a limited input subset. However, these are detected automatically and their verbose form could easily be transformed using a more compact notation of the following form:

input (* - {changed subelements}) = output (* - { corresponding subelements}).

Detecting modifications is traditionally associated to shape analyses, that focus on deep-heap mutations. Side-effect analyses detect memory locations that may be modified by an operation. We, however, are interested by deep-state modifications in the context of a functional language. Other analyses inferring frame properties have been devised. These are mostly used in a purely automatic setting. We however, developed a correlation analysis meant to be used in an interactive verification context.

Similarly to the case of the dependency analysis presented in Chapter 5, we have implemented a prototype of the correlation analysis in OCaml and we have applied it to a functional specification of ProvenCore (Lescuyer, 2015). Medium-sized experiments

performed on the abstract layers of ProvenCore show encouraging results. For instance, the correlation results of approximately 630 αSmil predicates, totalling approximately 10000 lines of code are obtained in less than 0.5 seconds, i.e. faster than the dependency summaries are obtained on the same predicates. This is partly a consequence of the fact that, unlike the dependency analysis which computes summaries for both code and specifications, the correlation analysis computes non-trivial results only for code. Specifications are predicates with Boolean exit labels which generate no outputs. Since our correlation analysis computes fine-grained relations between parts of the inputs and parts of the outputs, it cannot detect anything non-trivial in their case. However, this would change if we were to extend our correlation analysis and track relations between parts of the inputs as well. This is a direction that we plan to investigate in the future. We will focus on the implementation and the discussion of the obtained results in Chapter 8. The prototype can be tested on the web page[3] dedicated to our correlation analysis, where multiple examples are provided and explained. Additionally, users can devise and test their own examples.

The correlation analysis presented in this chapter has been the subject of a previous publication (Andreescu, Jensen, and Lescuyer, 2016).

[3]Correlation Analysis Web Page: http://www.ajl-demo.fr/2016/

Chapter 8

Implementation, Application and Results

> Any fact becomes important when it's connected to another.
>
> Umberto Eco

In this chapter, we focus mainly on the practical aspects regarding our static analyses and the approach to using their results for inferring the preservation of certain logical properties. In Section 8.1 and Section 8.2 we give a brief overview of the implementations of our dependency and correlation analyses, respectively. In Section 8.3, we succinctly present ProvenCore, one of the two microkernels developed at Prove & Run, and discuss, in terms of execution times and precision, the experiments we made on its functional specification. In Section 8.4 we describe the manner in which the summaries computed by our dependency and correlation analyses are meant to be combined and used for reasoning about the preservation of certain logical invariants. We illustrate this approach and discuss it on some examples inspired by ProvenCore.

8.1 Implementation of the Dependency Analysis

Prototypes for both of our static analyses, the dependency analysis presented in Chapter 5 and its extension with symbolic dependencies presented in Chapter 6, as well as the correlation analysis presented in Chapter 7, have been implemented in OCaml (Rémy and Vouillon, 1997). While trying to retain close proximity to the analyses as presented theoretically, their implementation mildly diverges from them at certain points due to performance and scalability considerations. One of the main differences is related to the manner in which we store dependencies and partial equivalence relations. Based on the observation that in general, when considering complex transition systems, the states are characterized by properties depending only on a limited subset of their subelements, while most transitions modify only a limited subset of the input state's subelements, we adopt a more compact representation. This in turn is reflected in some of the operators as well.

8.1.1 Dependency Type and Operators

The *abstract dependency type* δ that mirrors the structure of associative arrays and algebraic data types was introduced in Chapter 5.2, on page 83. It is implemented by the recursive type `dep` shown below:

```
(** Implementation for the dependency type
    introduced in Chapter 5.2 **)
type dep =
  | Everything (*top*)
  | Impossible (*bottom*)
  | Nothing
  | Deferred of accesses (*symbolic*)
  | Struct of struct_typ * dep FMap.t
  | Variant of var_typ * dep CMap.t
  | Array of dep * (var * dep) option
```

The maps used for expressing dependencies for structures and variants use as keys fields and constructors, respectively:

```
type field
module FMap : EMap.S with type key = field

type cons
module CMap : EMap.S with type key = cons
```

In contrast to the extended abstract dependency type δ (Definition 6.4.1), the actual dependency for structures stores, in addition to the map associating dependencies to fields, the type `struct_typ` of the structure as well. Similarly, the actual dependency for variants stores the variant's type `var_typ` as well, in addition to the map associating dependencies to constructors.

As previously mentioned, we are targeting complex transition systems, such as operating systems and microkernels. In practice, transitions frequently map a large input state to a large output state, but for computing the output state they are concerned only with a limited subset of the input state. The number of subelements of a complex input on which the outcome of a predicate depends tends to be low compared to the total number of input subelements, so we are filtering fields mapped to $\oslash$, denoted by `Nothing` in our implemented dependency type, from dependencies for structures. Similarly, from dependencies for variants we are filtering constructors mapped to $\bot$, denoted by `Impossible` in our implemented dependency type.

As a consequence of this optimization, we need to know, and hence, store the types of structures and variants in order to correctly compare, join and reduce dependencies corresponding to such types. In addition, this is also useful for checking that the constructed dependencies are well-typed.

For building dependencies of the corresponding type we have implemented smart constructors. The dependency type is private and new dependencies can be constructed only by using the provided smart constructors.

As explained in Section 5.2, $\top$, $\oslash$ and $\bot$ can apply to any type. For instance, $\top$ can be seen as a placeholder for data that is needed in its entirety. Structure, array or variant dependencies whose subelements are all entirely needed and thus, uniformly mapped to $\top$, are transformed to $\top$. The $\bot$ dependency is a placeholder for data that cannot occur on a certain execution scenario. A whole variant value is impossible if all its constructors are mapped to $\bot$. A whole structure or array is impossible if any of its subelements is impossible. These canonizations[1] are made by our smart constructors. For instance, the smart constructor for structure dependencies returns `Everything` if it receives as an input a map of fields in which each key is mapped to `Everything`. Since fields that are absent from a field map must be interpreted as being mapped to `Nothing`, before returning `Everything`, the constructor also verifies that the map of fields it received as an input contains all the fields of the structure type `struct_typ` given as an input as well. If the given map of fields contains an `Impossible` value, the smart constructor returns `Impossible`. Any mapping `field` $\mapsto$ `Nothing` is filtered from the given input map.

Similarly, for variant dependencies, the corresponding smart constructor receives as inputs the variant's type and a map from constructor keys to dependency values. If all constructors of the variant, as indicated by its type `var_typ`, are present in the input map and mapped to `Everything`, the smart constructor returns `Everything`. If all constructors are present and mapped to `Impossible`, the smart constructor returns `Impossible`. Otherwise, if the input map contains some constructors mapped to `Impossible`, the corresponding mappings are filtered from the map used to build the variant dependency.

For arrays, the smart constructor returns `Everything` if both the default dependency and the known exceptional dependency are `Everything` or if the former is `Everything` and there is no known exceptional dependency. If any of the two dependencies is `Impossible`, the smart constructor returns `Impossible`.

The smart constructor for deferred dependencies receives a set of variables as an input. If the given set is empty, the constructor returns `Nothing`. Otherwise, it creates the access map having the variables in the given input set, i.e. the root variables for symbolic paths, as keys. As described in Section 6.5, a set containing a single path, the *empty* path, is initially associated to each.

The $\sqsubseteq$ operator (Definition 5.2.2) as formally presented in Section 5.2 and detailed in Table 5.1 on page 86, returns *false* whenever comparing two incompatible dependencies. In practice, situations in which comparisons on incompatible types are made, should never be reached. As a consequence, whenever we compare structure or variant dependencies, we check, as a safety measure, that the two dependencies correspond to structures or variants of the same type. Otherwise, the two dependencies are not

[1] For making all the described canonizations, we have to make sure that whenever we replace δ by δ', both $\delta \sqsubseteq \delta'$ and $\delta' \sqsubseteq \delta$ hold.

comparable and we throw an exception that indicates that the types are incompatible. For structure dependencies, whenever a mapping for one field `f` can be found only in one of the two maps to be compared, we compare its mapped dependency value to `Nothing`, since absent fields must be interpreted as being mapped to `Nothing`. Similarly, for variant dependencies, whenever a mapping for a constructor `C` can be found only in one of the two maps to be compared, we interpret it as being mapped to `Impossible`.

The join (Definition 5.2.3) and reduction operator (Definition 5.2.4) as formally presented in Section 5.2 on page 87 and 89, respectively, are *total*: they return $\top$, the element conveying no information, for incompatible dependencies. In practice, the two operators are *partial*: an exception is thrown whenever the two dependencies to be joined or reduced are incompatible. This applies to structures or variant dependencies that do not correspond to the same type as well. Otherwise, when joining or reducing two compatible structure or variant dependencies, we interpret missing fields or missing constructors as being mapped to `Nothing` or `Impossible`, respectively.

In Section 6.6.1, we described that there are two types of free variables that can appear in dependencies. The first type consists of *index variables* that can appear in array dependencies. For instance, in `<Nothing ^ i : Everything>`, the variable `i` is the index of the cell for which the exceptional dependency `Everything` is known. Additionally, such index variables can also appear in symbolic paths related to arrays, such as: `<Nothing ^ i : Deferred(a[i])>` or `<Deferred(a[* - i]) ^ i : Nothing>`. Such indices must be input variables of the currently analysed predicate as explained in Section 5.3.2, on page 97. The second type of free variables are the *root variables* that appear in deferred dependencies. For instance, in `<Deferred(a[* - i]) ^ i : Nothing>` the variable `a` is a root variable. In the general case, the root variables are those outputs to which symbolic access paths are associated in deferred dependencies. In order to make use of the computed context-sensitive information, actual dependencies can be substituted for the root variables. This is done by applying the symbolic access paths to the dependency to substitute. By traversing entire dependencies such as:

```
{ f -> <Nothing ^ j : Everything>;
  g -> { b -> Deferred(o);};
  h -> { x -> Everything;
         y -> <Deferred(a[* - j]) ^ j : Nothing>;}
```

and substituting the nested deferred dependencies such as `Deferred(a[* - j])` and `Deferred(o)`, we apply context-sensitive information. Simultaneously, during the same traversal, we also substitute the indices appearing in array dependencies, such as `j` in the dependency associated to the field `f`, for instance. These are either substituted by another index variable or they are *forgotten*. If the index to substitute is an input, the formal variable will be replaced by the effective one. Otherwise, an approximation is made, in order to remove the local index variable. This consists in joining the default and the exceptional dependencies and using the result for building a new array dependency without an exception.

An index substitution is a mapping from variables to either a new index variable to replace it, or to `Forget`, if all references to the index variable should be removed. The index type is shown below:

```
type index = | NewIdx of var | Forget
```

The substitution function `subst` has the following type:

```
type var
module VMap : EMap.S with type key = var

val subst : index VMap.t -> dep VMap.t -> dep -> dep
```

Its first argument is the index substitution; the second argument is the dependency substitution mapping root variables to dependencies. The third argument is the dependency on which the substitutions are to be made. The function returns the dependency obtained after making both substitutions. The two substitution passes are fused for performance considerations.

A separate substitution is performed for dealing with polymorphic types. Our dependency type is not polymorphic per se. However, αSmil supports polymorphic types and thus, the variables described by the computed dependencies can have a polymorphic type. Since the types of structures and variants are stored in the corresponding dependencies, we must substitute polymorphic type parameters by their effective arguments. This is done by a recursive function which traverses the dependencies and makes the type substitution at each nested level if necessary. Besides this substitution, no other modifications were made in the implementation in order to handle polymorphism. This justifies our formal presentation of the analyses without polymorphism.

8.1.2 Intraprocedural Dependency Analysis

The *intraprocedural dependency type* Δ (Definition 5.3.1) mapping variables to dependencies δ that was introduced in Chapter 5.3.1 is implemented as shown below:

```
type reachable = dep VMap.t

(** Implementation of the intraprocedural dependency domain
    introduced in Chapter 5.3.1 **)
type intra =
  | Unreachable
  | Reachable of reachable
```

The `VMap` type is a map having variables as keys:

```
type var
module VMap : EMap.S with type key = var
```

In order to avoid needlessly storing large maps predominantly containing variables mapped to `Nothing`, we do not store by default mappings for variables for which dependencies have not yet been computed. Therefore, the intraprocedural dependency of any variable `v` for which a mapping has not yet been stored in the map is interpreted as $\texttt{v} \mapsto \texttt{Nothing}$. As discussed in the previous section for the partial order, join, and reduction operators, when applying $\sqsubseteq_\Delta$ (Definition 5.3.3), and the join $\vee_\Delta$ (Definition 5.3.4) and reduction $\oplus_\Delta$ (Definition 5.3.5) operators at the intraprocedural level any missing mapping from a `Reachable` domain has to be interpreted as a variable mapped to `Nothing`.

With this interpretation, *forgetting* a variable `v` (Definition 5.3.2) from an intraprocedural domain, denoted by $\setminus$ in Chapter 5.3.1, becomes straightforward and amounts to simply removing the mapping for `v` from the intraprocedural domain:

```
(* Forget *)
let forget d v =
  match d with
    | Unreachable -> d
    | Reachable dmap -> Reachable (VMap.remove v dmap)
```

We remark that the complex operations are performed at the dependency type level, and are mostly applied pointwise at the intraprocedural level. The interprocedural dependency domains are mappings from labels to intraprocedural dependency summaries.

8.2 Implementation of the Correlation Analysis

8.2.1 Partial Equivalence Relations and Operators

The *partial equivalence type* R (Definition 7.2.1) that mirrors the structure of associative arrays and algebraic data types which was introduced in Chapter 7.2.1 on page 141 is implemented as shown below:

```
(** Implementation of the partial equivalence type
    introduced in Chapter 7.2 **)
type pequiv =
  | Equal (*bottom*)
  | Any   (*top*)
  | PStruct of struct_typ * pequiv FMap.t    (*structures*)
  | PVariant of var_typ * pequiv CMap.t  (*variants*)
  | PArray of pequiv * (var * pequiv) option (*arrays*)
```

The `FMap` and `CMap` types are the ones presented on page 174.

Similarly to structure and variant dependencies, and due to the same practical considerations, in addition to the map associating partial equivalences to fields, the

type `struct_typ` of the structure is stored as well. Similarly, the implemented partial equivalence for variants stores the variant's type `var_typ` as well, in addition to the map associating partial equivalences to constructors.

For avoiding to store large maps in which the majority of the fields or constructors are mapped to `Any`, we filter mappings of the type `field` $\mapsto$ `Any` and `cons` $\mapsto$ `Any`.

The partial equivalence type is private and the only manner in which partial equivalence relations can be built is by using the provided smart constructors. The two atomic cases `Equal` and `Any`, respectively can apply to any type. The smart constructors for partial equivalences corresponding to structures filters out any field mapped to `Any`. It also returns `Equal` if all fields of the structure are mapped to `Equal` in the given input map. If, on the contrary, the given input map is empty or all fields are mapped to `Any`, the smart constructor returns `Any`.

Similarly, for partial equivalences corresponding to variants, the corresponding smart constructor receives as inputs the variant's type and a map with constructor keys and partial equivalences. If all constructors of the variants, as indicated by their type, are present in the input map and mapped to `Equal`, the smart constructor returns `Equal`. If all constructors are present and mapped to `Any` or if the given input map is *empty*, the smart constructor returns `Any`. Otherwise, if the input map contains some constructors mapped to `Any`, the corresponding mappings are filtered from the map used to build the variant partial equivalence.

For arrays, the smart constructor returns `Equal` if both the default relation and the known exceptional relation are `Equal` or if the former is `Equal` and there is no known exceptional relation. If both the default relation and the known exceptional relation are `Any` or if the former is `Any` and there is no known exceptional relation, the smart constructor returns `Any`.

In contrast to dependencies, there is only one type of free variables that can appear in partial equivalence relations, namely *index variables*. As was the case for array dependencies, these can appear in partial equivalence relations corresponding to arrays and they must be input variables. We traverse the partial equivalences recursively, checking for each index variable appearing in an array relation if it is an input or a local variable. References to local variables are eliminated, by approximating the partial equivalences, effectively joining the default array relations with the exceptional array relations.

8.2.2 Intraprocedural Correlations

In Chapter 7.4 on page 156 we have defined intraprocedural correlation summaries (Definition 7.4.1) as mappings from pairs of variables to correlation maps. In practice, the type `intra` is the following:

```
module PVMap = EMap.Make
  (struct type t = element * element let compare = compare end)

module PMap = EMap.Make
  (struct type t = Path.t * Path.t let compare = compare end)
```

```
type correlation = pequiv PMap.t
type intra = correlation PVMap.t

type t =
  | Related of intra
  | NoCorrelation
  | Unreachable
```

The implemented intraprocedural correlation summary type `intra` is a mapping from pairs of *elements* to correlation maps. The element type is shown below:

```
(** The type of the elements for which correlations
    are computed and kept intraprocedurally.
    Ghost elements are used only for variants: for a
    variant [v], a ghost element that nests the type
    of the variant [v] is created. These are filtered
    from final results. **)
type element =
  | Local of var
  | Output of var
  | Ghost of texpr
```

In practice, we need to distinguish between output variables and local variables. This is important for distinguishing between the final value of an output, i.e. the one correlated with values of the inputs, and its local intermediate values. Furthermore, we need to introduce *ghost* elements for variants. When constructing a variant `v` with a constructor `C(a,b)` for instance, we can keep correlations between the pairs `(a,v)` and `(b,v)`. However, we fail to capture the information regarding `v`'s construction with `C`. In order to maintain it, we create a ghost element `g_vtyp` with `v`'s type, we add the pair `(g_vtyp,v)` to the intraprocedural summary, and associate $(\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto [\texttt{C} \mapsto \mathsf{Any}]$ to it. Such pairs are deleted from the intraprocedural predicate summaries; they are only used while analysing a predicate's body.

Unlike the operations discussed in Chapter 7, the implementations of the partial order (Definition 7.4.2) and join (Definition 7.4.3) operations are parameterized by the typing environment mapping variables to types. This has to be threaded through all operations, as it is necessary for the injection operation (Definition 7.3.8). We need to know the variable type onto which the relation is injected. For instance, in order to "fill" the unknown relations for fields or constructors with Any, we must first know what those fields or constructors are.

8.2.3 Dependency and Correlation Analysers

The input program is first parsed and each predicate is analysed in turn. Implicit predicates are treated conservatively. Since their implementation is hidden, a pessimistic assumption must be made. For the dependency analysis it is considered that everything in their inputs has been read in order to obtain the outputs for any possible exit

label. Similarly, for the correlation analysis it is considered that there is no correlation between the input and the output variables on any possible exit label.

For inductive predicates, the dependency analysis computes a summary for each case and joins the results for obtaining the dependency summary for the **true** exit label. The **false** label is treated conservatively and everything is considered to be read. Since inductive predicates are specification-only predicates that do not generate outputs, the correlation analysis associates a NoCorrelation summary to both labels.

```
(** Analyse the body [g] of an explicit predicate **)
let analyze g =
  let todo = Queue.create () in
  List.iter (fun v -> Queue.push v todo) (G.vertices g);
  let result = init_result g in
  let rec progress r =
    try
      let v = Queue.pop todo in
      let vd = MV.find v r in
      let edges = preds g v in
      let vd' = transfer r v edges in
      if D.leq vd' vd then progress r
      else begin
        List.iter (fun edge ->
                     Queue.push (source edge) todo) edges;
        progress (MV.add v (D.join vd vd') r)
      end
    with Queue.Empty -> r
  in
  progress result
```

The body of each explicit predicate is analysed independently for each possible exit label using a variation of the worklist algorithm, as shown above in the `analyze` function. Initially, a map is created having as many elements as there are nodes in the predicate's body. All of these are initially mapped to `Unreachable`, the bottom element at the intraprocedural level. All the predicate's exit nodes are loaded into the working queue. Then a recursive function `progress` is executed until a fixed point is reached and there are no more nodes left to analyse in the working queue. The first node of the queue is popped and analysed. The node's summary as stored in the map is retrieved in `vd`. The analysis returns a summary `vd'` for the node. The two summaries, `vd'` and `vd` are compared, and if the former is more precise than the latter, then the recursive function `progress` is called. Otherwise, before calling `progress`, the predecessors of the analysed node are pushed into the working queue and, in the map of nodes, the join of `vd` and `vd'` is associated to the analysed node. Since both analyses are backwards analyses, the dependency and correlation information of a node is based on the dependency or correlation information of its successors in the control flow graph and the former must be recomputed if the latter are modified. Finally, from the computed intraprocedural dependency summary, all mappings corresponding to local variables

are filtered. From the computed correlation summary of an exit label `l`, all mappings that do not correspond to an input and output variable pair are filtered.

For the dependency analyser, a command-line flag can be used to disable the usage of deferred dependencies. Also, the well-typedness check of dependency summaries can be enabled similarly.

A parser for dependency information has been implemented as well. This allows us to annotate αSmil programs with the expected results and compare them to the computed ones. A similar parser for the correlation information is planned for the near future.

8.3 Dependency and Correlation Results on ProvenCore Layers

8.3.1 ProvenCore Description

ProvenCore (Lescuyer, 2015) is one of the two microkernels entirely specified and developed in Smart at Prove & Run. Unlike Minix 3.1 by which it was inspired, ProvenCore targets ARM architectures and uses a Memory Management Unit for managing virtual address spaces. It is a general-purpose microkernel supporting creation and deletion of processes, execution of programs, synchronous message-passing inter-process communication with timeouts, asynchronous notifications, and process-to-process data copies.

The main property ensured by ProvenCore is the isolation property. Isolation implies two complementary properties, namely integrity and confidentiality. Integrity refers to ensuring that the resources of a process (its code, data, and registers) cannot be altered or interfered with by other processes, unless explicitly authorized by the process. Confidentiality refers to ensuring that the resources of a process cannot be observed by other processes, unless explicitly authorized by the process. In other words, integrity ensures that until a process decides to communicate with other processes, it will execute as if it were alone on the system. Confidentiality ensures that as long as a process does not send its secrets to other processes, it can change its secrets without affecting other processes.

The isolation property has been formally proven using the interactive proof assistant of ProvenTools. The proofs also establish functional specifications verified by ProvenCore (Lescuyer, 2015).

The proof for the isolation property is based on multiple refinements between successive models, from the most abstract, on which the *isolation* property is defined and proven, to the most concrete, i.e. the actual model used for code generation. These successive models are shown in Figure 8.1.

Using multiple abstract models, each more abstract than its predecessor, enables a degree of separation of concerns in the overall proof. The lower-level proofs include a plethora of low-level properties and invariants, and are devoid of functional properties, while the higher-level models focus on functional specifications. Each layer of abstraction removes details that are not relevant for it anymore and enables changing

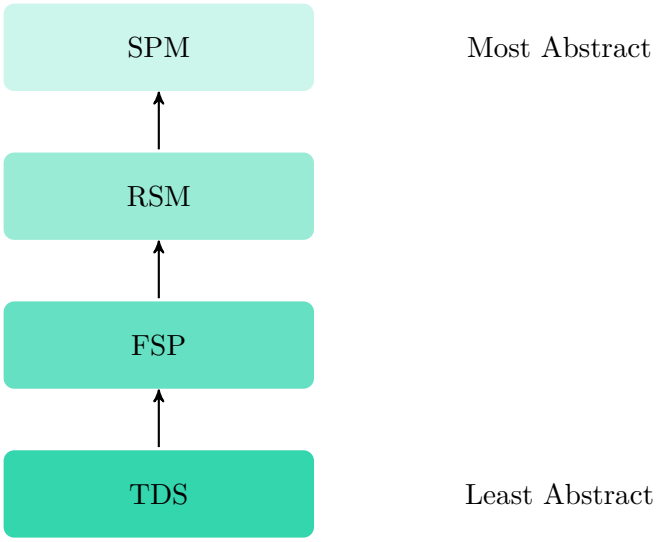


FIGURE 8.1 – ProvenCore – Abstract Layers

the representation of the transition system in order to internalize in the structure of its states some invariants of the preceding level.

The *Security Policy Model* (SPM) is the most abstract level and the one at which the isolation property is expressed and proven. The kernel is modeled as an abstract controller and the various processes are modeled as machines, each possessing its own independent physical resources.

The *Refined Security Model* (RSM) is an intermediate layer, meant to bridge the wide gap between its successor, the SPM and its predecessor, the FSP. In the RSM, the machines share the same physical resources, which are managed by the controller.

The *Functional Specifications* (FSP) layer is a model roughly equivalent to its predecessor – the TDS – in functionality, but unlike the latter it uses data structures and algorithms that facilitate reasoning and formal proof. Its main functional difference with the TDS is that it eliminates MMU address translation, using instead a linear view of the RAM, similarly to the RSM.

The *Target of Evaluation Design* (TDS) is the model that is used to generate the sequential Smart code of the kernel, as well as the models for hardware components that are not translated into C code, but which are necessary for completing the TDS specifications.

For each refinement, a *view*, i.e. a function from the concrete model state to the abstract model state, is defined. Then, a correspondence or commutation lemma is proven, establishing that transitions from c to c' in the concrete model entail transitions from the view of c to the view of c' in the abstract model. Since the views are not total functions, this requires showing that the views actually exist. In this manner, the higher levels are attained, reaching models that are simpler and more flexible than the TDS but that still simulate all its possible behaviours (Lescuyer, 2015).

This refinement chain also facilitates reusing parts of one proof effort in other proofs.

8.3.2 Obtained Dependency and Correlation Results

Our dependency and correlation analyses must be evaluated by two different criteria, namely execution time and precision. In this section we are discussing the former. The latter will be discussed in the following section.

Both analyses target complex transition systems in general, and operating systems in particular. The ideas behind them stemmed directly from the verification effort entailed by ProvenCore. Unlike other static analyses, which are frequently employed in a fully automatic setting, our static analyses are supposed to be used as companion tools in the middle of interactive program verification. They are supposed to be applied often, as steps during interactive proofs. For instance, the dependency and correlation summaries for different predicates might be needed for verifying a single property. These in turn may imply a whole-model analysis. Therefore, the dependency and correlation analyses must perform quickly in order to answer effectively "questions" asked frequently.

Our analyses have currently been applied to the functional specification of ProvenCore (Lescuyer, 2015). More specifically, they have been applied to the RSM, FSP, and TDS layers shown in Figure 8.1. Each of these layers is characterized by a global state with numerous fields, and different transitions, i.e. supported commands or system calls such as *fork, exec, exit*. Each supported command receives as an input the global state before the transition and returns the state of the system after the transition.

For instance, in RSM the global states are much simpler compared to the ones in the layers below it, i.e. FSP and TDS. They are modeled by a structure with 6 fields, out of which 3 are modeled by arrays and 2 by structures. The RSM counterpart of the optional table of processes is a store of machines, which are themselves the counterpart of FSP processes. Machines are structures with 7 fields that refer to registers, information regarding inter-process communication or permissions, and code and data segments. Out of the 7 fields, 2 are modeled by variants, 2 by associative arrays and other 2 by structures.

The global state of the FSP layer is modeled by a structure type with 15 fields, including fields that concern process management (for memory allocations, information about processes), interrupt handling (registered handlers, active handlers), scheduling (priority queues, currently running process, process to run next), time management or code data. Among these 15 fields, 9 fields are "composite" themselves, being modeled by structures, variants or associative arrays. For instance, among the fields concerning process management, there is a table of optional processes. The processes themselves are modeled by a structure type having 26 fields. Out of the total of 26 fields, 11 are modeled by algebraic data structures or associative arrays too.

The FSP global state is characterized by over 70 invariants.

In TDS, the global state is a structure having 33 fields, among which 23 are "composite" as well. The processes are structures having 29 fields, among which 14 are modeled by associative arrays or algebraic data types. The global state is characterized by approximately 140 invariants.

In Table 8.3 we give an overview of the global states for each analysed layer. The first column shows the total number of fields. The second column indicates the number of fields that are modeled by associative arrays. Between parentheses we indicate the number of arrays having "composite" elements and elements of atomic or implicit types, respectively. For example, the FSP global state has 6 fields that are modeled by associative arrays and all 6 of them have "composite" elements. In columns 3, 4, and 5 we show the number of fields that are modeled by structures, variants and atomic or implicit types, respectively.

Table 8.3 – ProvenCore Abstract Layers – Global State Type

	Global State	Arrays	Structures	Variants	Atomic/Implicit
RSM	6 fields	2 fields (1/1)	2 fields	0 fields	2 fields
FSP	15 fields	6 fields (6/0)	0 fields	3 fields	6 fields
TDS	33 fields	14 fields (14/0)	3 fields	6 fields	10 fields

The global state of each layer contains an array or store of processes or machines. In Table 8.4 we give an overview of the process or machine type for each analysed layer. The table has the same structure as the one described previously for the global state types.

Table 8.4 – ProvenCore Abstract Layers – Process/Machine Type

	Process/Machine	Arrays	Structures	Variants	Atomic/Implicit
RSM	7 fields	2 fields (1/1)	2 fields	2 fields	1 field
FSP	26 fields	2 fields (0/2)	5 fields	3 fields	16 fields
TDS	29 fields	1 field (1/0)	8 fields	5 fields	15 fields

We have applied our dependency and correlation analyses on the RSM, FSP, and TDS layers, thus conducting medium-sized experiments. An overview of the characteristics for the 3 ProvenCore layers is included in Table 8.5, Table 8.7, and Table 8.9. In each of these, the first column shows the total number of predicates of the analysed layers. In parentheses, we indicate the number of predicates that only read information and return a Boolean-like exit label, i.e. logical properties, as well as the number of implicit predicates for which a pessimistic assumption is made. The second column shows the total number of lines of code (LoC) for each, including comments and type definitions. The next three columns indicate the number of LoC corresponding to predicates, type definitions and comments, respectively.

We have run the analyses 101 times in a loop on a Lenovo laptop with a Quad-Core Intel Core I7-5500U processor and 8 GB RAM. The system runs Xubuntu Gnu/Linux 64 bit, Release 15.10 with OCaml 4.01. Before the first run of each loop the operating system's cache was dropped using the following command:

```
echo 3 > /proc/sys/vm/drop_caches
```

The time measured includes only the execution of the analysis algorithms. It excludes the time required to load the input files, as well as the time spent printing the results.

On average, our fully context-insensitive dependency analysis as presented in Chapter 5 computed the dependency summaries for 633 RSM/FSP predicates in 0.656 seconds. For the TDS predicates, the dependency summaries were computed in 0.699 seconds on average. These results are indicated in Table 8.5.

Table 8.5 – Abstract Layers – Evaluation Data and Dependency Analysis Timing

	Predicates	Total LoC	Code	Types	Comments	Dependency Avg
RSM/FSP	633 (235/65)	9853	8402	596	855	0.656 s
TDS	780 (231/155)	14000	11306	588	2106	0.699 s

In Table 8.6 we indicate the minimum and maximum execution times for the context-insensitive dependency analysis. Various percentiles are indicated as well.

Table 8.6 – Abstract Layers – Detailed Dependency Analysis Timing (in seconds)

	Min	10%ile	50%ile	90%ile	Max	Avg
RSM/FSP	0.650	0.651	0.652	0.658	0.730	0.656
TDS	0.690	0.691	0.693	0.718	0.798	0.699

The average execution time of our dependency analysis with the deferred accesses extension is shown in Table 8.7, in the last column denoted by Avg. On average, our dependency analysis extended with deferred accesses, as presented in Chapter 6, computed the dependency summaries with context-sensitive leaves for 633 predicates in 0.779 seconds. For the TDS predicates, the dependency information was computed in 0.919 seconds on average. These results are indicated in Table 8.7.

Therefore, using our relaxed form of context-sensitivity led to an increase of 10-20% in execution time on the used benchmarks.

The detailed timing information for the dependency analysis using deferred accesses is shown in Table 8.8.

The average execution time of our correlation analysis is shown in Table 8.9, in the last column denoted by Avg. The correlation summaries for the RSM/FSP predicates are computed in 0.426 seconds on average. For the TDS predicates, the correlation summaries are computed in 0.496 seconds on average. Unlike the dependency analysis

Table 8.7 – Abstract Layers – Evaluation Data and Deferred Dependency Analysis Timing

	Predicates	Total LoC	Code	Types	Comments	Deferred Avg
RSM/FSP	633 (235/65)	9853	8402	596	855	0.779 s
TDS	780 (231/155)	14000	11306	588	2106	0.919 s

Table 8.8 – Abstract Layers – Detailed Deferred Dependency Analysis Timing (in seconds)

	Min	10%ile	50%ile	90%ile	Max	Avg
RSM/FSP	0.776	0.777	0.779	0.781	0.785	0.779
TDS	0.904	0.905	0.908	0.975	0.999	0.919

which computes information for code as well as specifications, i.e. logical properties, in a unified manner, the correlation analysis only computes information for predicates that actually modify data structures. This partly explains the time difference between the two analyses. We also remark that the possible-constructors analysis is performed simultaneously with the dependency analysis and this contributes to the difference between the execution times as well.

Table 8.9 – Abstract Layers – Evaluation Data and Correlation Analysis Timing

	Predicates	Total LoC	Code	Types	Comments	Correlation Avg
RSM/FSP	633 (235/65)	9853	8402	596	855	0.426 s
TDS	780 (231/155)	14000	11306	588	2106	0.496 s

The detailed timing information for our correlation analysis is shown in Table 8.10.

Generally, static analysis has been considered prohibitive in terms of execution time and it has been avoided in an interactive context and used predominantly in an automatic context. Though currently applied only on medium-sized models, the execution times of both of our analyses are short enough to expect reasonable execution times for larger models as well[2].

[2]It is noteworthy to remark that the interprocedural dependency and correlation summaries will not necessarily be computed on-the-fly, during the interactive proof. They rather will be computed as part of the build. In contrast, the treatment of a query, once all interprocedural information has been

TABLE 8.10 – Abstract Layers – Detailed Correlation Analysis Timing (in seconds)

	Min	10%ile	50%ile	90%ile	Max	Avg
RSM/FSP	0.424	0.425	0.425	0.427	0.432	0.426
TDS	0.492	0.493	0.494	0.498	0.540	0.496

8.3.3 Precision of our Dependency and Correlation Summaries

In this section we try to illustrate the sort of dependency and correlation summaries that are computed by our analyses. We conclude the section with a brief discussion regarding the precision of our obtained results. Assessing and discussing precision as a metric for usefulness is hard in isolation and can only be effectively done in relation to actual applications. However, we present some statistics in order to give some insight about the proportion of the non-trivial information computed. For our current discussion we focus on the results obtained on the RSM/FSP and the TDS layers.

One of the analysed predicates of the RSM/FSP layers is `do_auth`. This predicate is a system call clearing or granting an authorization to some process to read from or write to some memory range of the current process. It receives a global state `in` and an index `i` as inputs and produces, on the **true** label, the new global state `out`, after modifying the permission for the `i`-th process in the process store.

The code of `do_auth` performs various system-wide checks before registering the permission change, and is therefore not trivial, although its effect is quite limited. Indeed, the correlation results computed by our analysis for the **true** label of this predicate are shown below.

```
true : (in, out) ↦ [
        (ε̂, ε̂)  ↦ { ...      ↦ Equal          } 14 fields
                   procs     ↦ Any }
(.procs, .procs) ↦ ⟨ Equal ▷ i :  [ None ↦ Equal
                                    Some ↦ {v ↦ { ...      ↦ Equal } 25
                                                                 fields
                                                 mem_auth ↦ Any}}]⟩]
```

The analysis detects that out of the 15 fields of `out`, only the `i`-th element of the `procs` field is changed. Furthermore, it detects that if this element is an active process, i.e. built with the `Some` constructor, only the `mem_auth` field is modified out of the total of 26 fields. Everything else is copied from the input state `in`.

computed, will be executed in real-time. Nevertheless, it is desirable to have fast analyses, allowing developers to iterate frequently.

Combined with dependency summaries for logical properties, this correlation summary would allow us to infer the preservation of all invariants that are not concerned with the memory permissions. All but one out of the specified properties for the global state fall into this category. This is the *relevant memory permissions* property:

```
predicate proc_mem_auth_ok(proc proc) -> [true | false]
```

which verifies a fundamental property that has to hold for all processes in the process store of `proc` and states that a process has permissions covering a valid range of memory addresses and referring only to existing processes. After executing `do_auth`, this property is threatened and needs to be verified only for the `i`-th process of the store. It is preserved for all others.

The dependency results computed by our analysis for this predicate are shown below. The analysis detects that for each of the possible execution scenarios, the outcome depends only on 2 out of the 26 fields, namely the stackframe and the memory permissions. The dependency on the `stackframe` is confined to only one of the 3 fields: the data and stack segment. The memory permissions are given by a variant with 3 constructors, denoting reading and writing permissions or the absence of any permission. Furthermore, besides pinning down the outcome's dependency on 2 out of the 26 fields of the `proc` structure, the analysis also detects that the absence of any memory permission, indicated by the constructor `NONE` of the `mem_auth` variant, is $\bot$ for the false execution scenario. In other words, unused permissions cannot threaten the property `proc_mem_auth_ok`.

$$
\begin{array}{rrrrl}
\textbf{false} \to \{\mathit{proc} \to \{ & \mathit{mem\_auth} \to [& \mathit{READ} & \to \{ & \mathit{base} \to \top;\ \mathit{len} \to \top\} \\
 & & \mathit{WRITE} & \to \{ & \mathit{base} \to \top;\ \mathit{len} \to \top\} \\
 & & \mathit{NONE} & \to & \bot] \\
 & \mathit{stackframe} \to \{ & \mathit{ds} & \to & \top\}\}\} \\
\textbf{true} \to \{\mathit{proc} \to \{ & \mathit{mem\_auth} \to [& \mathit{READ} & \to \{ & \mathit{base} \to \top;\ \mathit{len} \to \top\} \\
 & & \mathit{WRITE} & \to \{ & \mathit{base} \to \top;\ \mathit{len} \to \top\} \\
 & & \mathit{NONE} & \to & \oslash] \\
 & \mathit{stackframe} \to \{ & \mathit{ds} & \to & \top\}\}\}
\end{array}
$$

The *relevant memory permissions* property is thus only threatened by transitions that add memory permissions or change a process' virtual space layout. Only 2 transitions out of the 25 belong to this category: `exec` which resets the process' segments, and `do_auth` which adds permissions and was discussed above. In particular, transitions deleting memory permissions do not impact the property since the absence of permissions, as shown by the dependency of the constructor `NONE` for the false label, is an impossible case when the property does not hold. This is one of the practical advantages of tracking constructor possibilities simultaneously and of extending the correlation analysis to track the evolution of constructors as well.

In the following, we briefly discuss our dependency summaries obtained on the RSM/FSP layer in terms of precision. An overview is given in Table 8.11. The first column refers to the fully context-insensitive dependency analysis as presented in Chapter 5. The second column refers to the dependency analysis extended with deferred

access maps as presented in Chapter 6. The first line indicates the total number of predicates, both implicit and explicit. The second line indicates the total number of implicit predicates for which we are obliged to make a pessimistic assumption and to consider everything needed, given that their implementation is hidden. The third line indicates the number of explicit predicates without inputs for which empty summaries are retrieved. Our dependency analysis detects the input subset that is read in order to obtain the output. In the case of predicates without inputs this subset is empty. Most explicit predicates without inputs correspond to wrapper predicates around calls to constructors that take no arguments. Since αSmil is an intermediate language, such predicates are automatically generated and do not necessarily correspond to programmer written predicates. The next line, line 4, indicates the number of predicates for which we obtain non-trivial information. By non-trivial information, we mean dependency summaries in which the dependency associated to at least one input variable is different than $\top$, i.e. Everything, the element conveying no information. With the context-insensitive dependency analysis, we obtain non-trivial results for 344 predicates. With the extended dependency we obtain non-trivial results for 403 predicates.

Table 8.11 – RSM/FSP Layers – Evaluation Data and Dependency Summaries

	Context-Insensitive	Deferred
Number of Total Predicates	633	633
Number of Implicit Predicates	65	65
No Inputs	26	26
Number of Non-Trivial Results	344	403
Number of Trivial-Results	289	230
• Implicit	65	65
• No Inputs	26	26
• Other	198	139
Predicates with Atomic Inputs	31	31
Completely Read	71	71
Overapproximation	96	37

The following line — line 5 — indicates the total number of predicates for which trivial results are obtained. These include the results for implicit predicates, as well as those for predicates without inputs. For the simple version of the dependency analysis we obtain 198 trivial results, excluding implicit predicates and predicates without inputs. For the extended dependency analysis we obtain trivial results for 139 predicates, excluding implicit predicates and predicates without inputs. Therefore, for the first version of the analysis, 49 trivial summaries are a consequence of context-insensitivity. The

next 3 lines refer to the 139 predicates for which trivial results are obtained with both versions of the dependency analysis: 31 of them correspond to predicates manipulating only inputs of atomic types, such as `int`. Such inputs are completely read and thus, the trivial results are justified and do not correspond to an over-approximation. Other 71 correspond to predicates making complex manipulations and actually reading all of their input, such as well-formedness checks. The last 37 trivial results are a consequence of over-approximations made by our analysis. The majority of them correspond to complex predicates, making multiple calls to other complex predicates and relying heavily on calls to implicit predicates, for which conservative assumptions are made. For the simple dependency analysis, other 46 trivial results are a result of over-approximations related to context-insensitivity.

An overview of the dependency results for the TDS layer is given in Table 8.12. The table follows the same structure as described for Table 8.11.

TABLE 8.12 – TDS Layer – Evaluation Data and Dependency Summaries

	Context-Insensitive	Deferred
Number of Total Predicates	780	780
Number of Implicit Predicates	155	155
No Inputs	15	15
Number of Non-Trivial Results	386	458
Number of Trivial-Results	394	322
• Implicit	155	155
• No Inputs	15	15
• Other	224	152
Predicates with Atomic Inputs	49	49
Completely Read	59	59
Overapproximation	116	44

We remark that with the deferred dependencies extension, we obtain more precise dependency summaries for 273 predicates of the RSM/FSP abstract layer. These constitute approximately 50% of the predicates in the used benchmark. For the TDS layer we obtain more precise results for 308 predicates using the deferred dependencies extension. These constitute approximately 50% of the predicates in the TDS layer for which non-trivial results can be obtained (i.e. excluding implicit predicates and those without inputs). The dependency summaries obtained with the extended analysis are considerably more detailed. For instance, just to give an intuition of the difference between the results obtained for the TDS layer, the file containing the results computed with the context-insensitive dependency analysis contains 7333 lines and its size

is 263.1 kB, while the file containing the results computed with the extended analysis contains 11547 lines and its size is 523.9 kB.

The statistics for the correlation analysis are shown in Table 8.13. Unlike the dependency analysis, which handles both logical properties and predicates generating outputs, the correlation analysis does not handle logical properties. It tracks fine-grained partial equivalences between parts of the input and parts of the output. Therefore, the number of RSM/FSP predicates for which we can obtain non-trivial results (i.e. at least one partial equivalence between an input (sub)element and an output (sub)element, on at least one exit label) is lower. Implicit predicates and specification-only predicates are mapped to NoCorrelation, the top element conveying no information. Out of the 307 predicates left, we obtain non-trivial results for 186 of them. The rest include predicates relying heavily on calls to implicit predicates. They also include complex system calls such as `fork` or `exec` and auxiliary operations which modify their input entirely.

TABLE 8.13 – RSM/FSP Layers – Evaluation Data and Correlation Summaries

	Correlation Analysis
Number of Total Predicates	633
Number of Implicit Predicates	65
Number of Logical Properties (No Outputs)	235
No Inputs	26
Number of Non-Trivial Results	186
Number of Trivial-Results	90
• Implicit	65
• No Inputs	26
• No Outputs	235
• Atomic/Implicit Inputs	31

An overview of the correlation results for the TDS layer is given in Table 8.14. The table follows the same structure as described for Table 8.13.

8.4 Reasoning about Framing using Correlations and Dependencies

8.4.1 A Decision Procedure

In general, reasoning about framing relies on the *frame rule*, which is commonly illustrated as follows:

$$\frac{\{P\}C\{Q\}}{\{P \wedge R\}C\{Q \wedge R\}}$$

Table 8.14 – TDS Layer – Evaluation Data and Correlation Summaries

	Correlation Analysis
Number of Total Predicates	780
Number of Implicit Predicates	155
Number of Logical Properties (No Outputs)	231
No Inputs	15
Number of Non-Trivial Results	235
Number of Trivial-Results	95
• Implicit	155
• No Inputs	15
• No Outputs	231
• Atomic/Implicit Inputs	49

The purpose of the frame rule is to enable local reasoning: a property R that holds for a state P, will continue to hold after executing a command C, provided that R reads only locations that are unmodified by C. The frame rule, also called the rule of constancy (Reynolds, 1981), applies in its original form to simple languages which do not use a heap. Separation logic addresses framing for heap-supporting languages.

In our case, the αSmil language with which we are working does not support mutation. Our work is not concerned with heap modifications but focuses on deep-state modifications. We handle predicates that receive a composite input state and construct a new composite output state, without altering the former. The new output state is constructed by copying the input state and modifying a subset of subelements.

In our context, the frame rule must be reinterpreted as follows: a property R is preserved by a predicate C receiving an input state P and constructing an output state Q, if the states P and Q agree on the subset on which the property R depends. In other words, a property is preserved by a predicate, if the latter only modifies subelements on which the property does not depend. Using the terminology used in separation logic, a property R is preserved by a predicate C if the footprint of C is disjoint from the footprint of R. However, we are not concerned with locations, but with subelements of large states modeled by algebraic data structures and arrays. Therefore, when reasoning about framing we need to check if the input subset modified by an operation is disjoint from the subset that properties are reading and depending on.

We have devised two static analyses for automatically computing the footprints of operations and properties. The dependency analysis detects the input subset on which the outcome of an operation or of a property relies. The correlation analysis detects the input subset that is modified by an operation in order to obtain the output. The results of the two analyses are meant to be used and combined by a decision procedure in order to automatically infer the preservation of frame properties.

The decision procedure has not been implemented yet, but based on preliminary

experiments we give an intuition about how the dependency and correlation summaries are meant to be unified, what type of queries could be answered, and the mechanism used for answering them.

Concretely, the decision procedure is meant to receive a sequence of *atoms*, one of which is a *query*. The query is to be answered based on the correlation summaries computed for the other atoms. Atoms are calls to built-in or user-defined predicates. Queries usually consist of a Boolean built-in statement, such as an equality check or a partial structure equality check for instance, or a call to a logical predicate, having `true` and `false` as exit labels and generating no outputs. In a nutshell, the dependency summary computed for the query would have to be transformed and interpreted as a set of correlations that are sufficient to answer affirmatively the given query. This should then be compared to the correlations computed for the atoms. The query can be answered affirmatively if the latter is less than or equal to the former.

We sketch the envisioned mechanism behind our decision procedure on a simple example, receiving 4 atoms. One of them is a query, as shown below:

```
type state = {
  f : int;
  g : int;
  h : int;
}
```

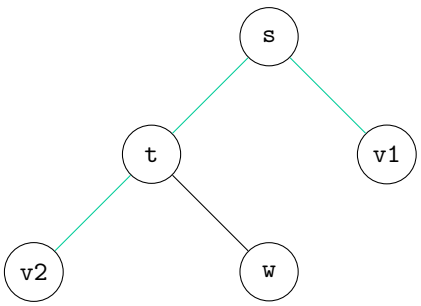
```
v1 := s.f;
 t := {s with g = w};
v2 := t.f;

Q: v1 = v2 - true -
```

In this case, it is not necessary to first obtain the dependency for the query marked with `Q` and to interpret it as a correlation. The necessary and sufficient correlation for the query to be answered affirmatively can be obtained directly:

$$(\texttt{v1}, \texttt{v2}) \quad \mapsto \quad \{(\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}\}.$$

Separately, we need to extract all the correlation information regarding (`v1`, `v2`) from the given atoms. For this, we must first find the chains of correlations connecting the two through other intermediate atoms. Therefore, we begin by building an undirected graph in which every variable appearing in the atoms is added as a node. An edge is added between any nodes representing the input and the output of the same atom[3]. For our example, the graph is shown below:



[3]In general, these graphs will not be acyclic. Further measures will have to be taken for correctly dealing with all cases.

The path connecting `v1` and `v2` is highlighted in green. In the general case, such paths could be detected using a depth-first search algorithm. Using the detected path between `v1` and `v2`, we build a chain of pairs of variables of the following form:

$$(\texttt{v1}, \texttt{s}) \quad \texttt{<->} \quad (\texttt{s}, \texttt{t}) \quad \texttt{<->} \quad (\texttt{t}, \texttt{v2}).$$

These are the *unordered* paths for which we need to extract the correlation information contained in the correlation summaries of the atoms. The correlation summaries of our example atoms are the following:

$$\texttt{v1 := s.f} \quad : \quad (\texttt{s}, \texttt{v1}) \mapsto \{(\texttt{.f}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}\}$$

$$\texttt{t := \{s with g = w\}} \quad : \quad \begin{array}{l} (\texttt{s}, \texttt{t}) \mapsto \left\{ \begin{array}{lll} (\texttt{.f}, \texttt{.f}) & \mapsto & \mathsf{Equal} \\ (\texttt{.h}, \texttt{.h}) & \mapsto & \mathsf{Equal} \end{array} \right\} \\ (\texttt{w}, \texttt{t}) \mapsto \{(\widehat{\varepsilon}, \texttt{.g}) \mapsto \mathsf{Equal}\} \end{array}$$

$$\texttt{v2 := t.f} \quad : \quad (\texttt{t}, \texttt{v2}) \mapsto \{(\texttt{.f}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}\}$$

In the correlation summaries computed by our analysis, correlation maps are associated to pairs of input and output values, i.e. the computed information is expressed between the input and the output variables of an operation. They can be seen as ordered pairs, having inputs as the left members and outputs as the right members. However, the correlation information expresses a relation between two runtime values which can be compared independently of the order in which they appear[4]. The atoms refer to values that occur in the program at different times, and answering the query is done independently of the order of execution. Therefore, at this level, we can swap the members of the pairs to which correlation maps are associated. This allows us to obtain correlation information expressed in terms of the variable pairs in the chain extracted from the graph of atom variables. For instance, for our example we would obtain the following:

$$(\texttt{v1}, \texttt{s}) \quad \texttt{<->} \quad (\texttt{s}, \texttt{t}) \quad \texttt{<->} \quad (\texttt{t}, \texttt{v2}).$$

$$(\texttt{v1}, \texttt{s}) \mapsto \{(\widehat{\varepsilon}, \texttt{.f}) \mapsto \mathsf{Equal}\}$$

$$(\texttt{s}, \texttt{t}) \mapsto \left\{ \begin{array}{lll} (\texttt{.f}, \texttt{.f}) & \mapsto & \mathsf{Equal} \\ (\texttt{.h}, \texttt{.h}) & \mapsto & \mathsf{Equal} \end{array} \right\}$$

$$(\texttt{t}, \texttt{v2}) \mapsto \{(\texttt{.f}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}\}$$

From these, we compute the Cartesian product of the correlations appearing in the correlation maps as follows:

[4] When the evolution of constructors will be tracked as well, the relations will stop being symmetric. Thus, the matrices will have to be transposed.

$$\{c_1\} \times \{c_2, c_3\} \times \{c_4\}$$

where

$$\begin{array}{lll} c_1 & = (\widehat{\varepsilon}, \texttt{.f}) & \mapsto \mathsf{Equal} \\ c_2 & = (\texttt{.f}, \texttt{.f}) & \mapsto \mathsf{Equal} \\ c_3 & = (\texttt{.h}, \texttt{.h}) & \mapsto \mathsf{Equal} \\ c_4 & = (\texttt{.f}, \widehat{\varepsilon}) & \mapsto \mathsf{Equal}. \end{array}$$

For our example, the obtained set would be the following:

$$\left\{ \begin{array}{lll} ((\widehat{\varepsilon}, \texttt{.f}) \mapsto \mathsf{Equal}; & (\texttt{.f}, \texttt{.f}) \mapsto \mathsf{Equal}; & (\texttt{.f}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}); \\ ((\widehat{\varepsilon}, \texttt{.f}) \mapsto \mathsf{Equal}; & (\texttt{.h}, \texttt{.h}) \mapsto \mathsf{Equal}; & (\texttt{.f}, \widehat{\varepsilon}) \mapsto \mathsf{Equal})) \end{array} \right\}$$

For each member of the obtained set we need to recursively *compose* the correlations in order to obtain information regarding the values involved in the query. The *compose* operations would be applied as follows:

$$(((c'_1 \odot c'_2) \odot c'_3) \odot \cdots)$$

where, for the first element of our example set, c'_1, c'_2 and c'_3 have the following values:

$$\begin{array}{lll} c'_1 & = (\widehat{\varepsilon}, \texttt{.f}) & \mapsto \mathsf{Equal} \\ c'_2 & = (\texttt{.f}, \texttt{.f}) & \mapsto \mathsf{Equal} \\ c'_3 & = (\texttt{.f}, \widehat{\varepsilon}) & \mapsto \mathsf{Equal}. \end{array}$$

For our example, we cannot obtain any correlation information regarding (`v1`, `v2`) by composing the correlations of the second member of the Cartesian product. The first correlation relates the value of `v1` to the value of the `f` field of `s`, while the second correlation relates the values of the field `h` of `s` and `t`. Thus, in this case we cannot infer anything regarding `v1` and `t`, nor regarding `v1` and `v2`. However, by composing the correlations of the first member of the Cartesian product, we obtain the following:

$$\left\{ \ (\texttt{v1}, \texttt{v2}) \mapsto (\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}; \ \right\}.$$

If after composing, we would have obtained multiple correlations referring to (`v1`, `v2`), these would have had to be intersected, thus allowing us to extract from the given atoms the most precise correlation information regarding (`v1`, `v2`). In the general case, the correlation information obtained after the intersection is the one that has to be compared to the correlation computed previously, i.e. the sufficient correlation for the query to be answered affirmatively. For our example this amounts thus to comparing:

$$\left\{ \ (\texttt{v1}, \texttt{v2}) \mapsto (\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}; \ \right\} \sqsubseteq_{\mathcal{K}} \left\{ \ (\texttt{v1}, \texttt{v2}) \mapsto (\widehat{\varepsilon}, \widehat{\varepsilon}) \mapsto \mathsf{Equal}; \ \right\} .$$

Based on this, we can conclude that the given query `Q` will be answered affirmatively for the atoms given in our example.

8.4.2 Types of Targeted Queries

The types of queries that are targeted by our approach can be categorized as follows:

- equality of values;
- structure equality on the values of a subset of fields;
- implications of the form $\mathsf{logical\_property}(\bar{a}) \Rightarrow \mathsf{logical\_property}(\bar{b})$, where $\bar{a}$ and $\bar{b}$ are related by the facts inferred from the other atoms of the query;
- conjunctions of such queries.

In the general case, we need to reinterpret a dependency summary as a correlation summary. The query's goal is to deduce the equality between pairs of variables. When two such variables are of the same type, we can create a correlation map containing a single correlation. That correlation associates to the pair of paths $(\hat{\varepsilon}, \hat{\varepsilon})$ a partial equivalence relation which mirrors the dependency. The partial equivalence relation is created as follows:

- When the dependency is Everything, the equivalence relation becomes Equal;
- When the dependency is Nothing, the equivalence relation becomes Any;
- Structure, variant and array dependencies are transformed pointwise to structure, variant and array partial relations;
- When the dependency is Impossible, the equivalence relation becomes Any, in the absence of the possible-constructors extension.

We illustrate here some example queries revolving around our `do_auth` predicate discussed in Section 8.3.3.

A naive equality query on the entire input and output of `do_auth` would not be satisfiable as `do_auth` does modify the memory authorizations of one process. This is the first sort of supported query.

```
do_auth(now, i, arg3, ...)[true: after|oob|false]
Q after = now
⇒ no.
```

The main argument of the `do_auth` predicate is the global state `now`, an instance of the `global_state` structure[5]:

```
type global_state = {
  procs: array<option<process>>;
  memory_regions: array<mem_region>;
  irq_handlers : array<irq_handler>;
  current_process: int;
  ...
}
```

[5]Due to confidentiality reasons, the actual definition of the struct has been modified and edited for length.

Since the `do_auth` predicate only affects the `mem_auth` of one process in the `procs` array, we can successfully deduce, for the values of `now` and `after`, the equality on the fields `memory_regions` and `current_process`. This is the second sort of supported query.

```
do_auth(now, arg2, arg3, ...)[true: after|oob|false]
Q after = <memory_regions current_process>now
⇒ yes.
```

Finally, we can directly deduce that the `all_ids_in_handlers_ok_global(state)` property is not threatened by the execution of the `do_auth` predicate.

```
do_auth(now, arg2, arg3, ...)[true: after|oob|false]
Q congruent all_ids_in_handlers_ok_global(now)
            all_ids_in_handlers_ok_global(after)
⇒ yes.
```

This property verifies that all the identifiers used by the registered interruption handlers stored in the field `irq_handlers` are valid. The property has the following dependency summary:

$$\begin{array}{lcl} \mathsf{false} & \to & \{\mathsf{state} \to \{\mathsf{irq\_handlers} \to \mathsf{Everything}\}\} \\ \mathsf{true} & \to & \{\mathsf{state} \to \{\mathsf{irq\_handlers} \to \mathsf{Everything}\}\} \end{array}$$

From the correlation of the `do_auth` predicate, we know that the `irq_handlers` field is preserved, and therefore it follows that the property, which only depends on that field is preserved. Similar properties that do not depend on the `procs` array, but only on parts or on the entirety of one or more of the other 14 fields will be preserved as well.

The preservation of properties that have to hold for every process in the array `procs` will be inferred as well, as long as they do not depend on the `mem_auth` field of the processes. For instance, the property `procs_proc_map_ok_global` verifies that each process of the array `procs` has valid code, data and stack segments. This property has the following dependency summary:

$$\mathsf{true} \to \left\{\mathsf{state} \to \left\{\mathsf{procs} \to \left\langle\left[\begin{array}{ll} \mathsf{None} & \to \mathsf{Everything}; \\ \mathsf{Some} & \to \{\mathsf{v} \to \{\mathsf{proc\_map} \to \mathsf{Everything}\}\} \end{array}\right]\right\rangle\right\}\right\}$$

$$\mathsf{false} \to \left\{\mathsf{state} \to \left\{\mathsf{procs} \to \left\langle\left[\begin{array}{ll} \mathsf{None} & \to \mathsf{Everything}; \\ \mathsf{Some} & \to \{\mathsf{v} \to \{\mathsf{proc\_map} \to \mathsf{Everything}\}\} \end{array}\right]\right\rangle\right\}\right\}$$

Since for every active process of the array, the property depends only on the `proc_map` field, it is unaffected by the modification of the `mem_auth` field. Therefore, the property is preserved for the global state `after` obtained after the execution of `do_auth`. Similar properties that do not depend on the `mem_auth` field, but only depend on other parts of the data structure will be preserved as well.

An extension of the decision procedure sketched in Section 8.4.1 could take advantage of additional information regarding array indices. For example, the query could specify that two of the involved array indices are different.

```
do_auth(now, i, arg3, ...)[true: after|oob|false]
Assert i != j
Q congruent mem_auth_ok_global(now, j)
            mem_auth_ok_global(after, j)
⇒ yes.
```

The `mem_auth_ok_global(state,j)` property checks the well-formedness of the memory permission on the `j`-th process. The above query is satisfied if the property `mem_auth_ok_global` holds for all processes other than the `i`-th. The correlation summary for `do_auth` states that the elements of the `procs` array are unmodified by the operation, except for the `i`-th element. Combined with the dependency summary for `mem_auth_ok_global` given below, this allows the query to be satisfied.

$$\mathsf{true} \to \left\{\mathsf{state} \to \left\{\mathsf{procs} \to \left\langle \mathsf{Nothing} \triangleright \mathsf{j} : \left[\begin{array}{ll} \mathsf{None} & \to \mathsf{Everything} \\ \mathsf{Some} & \to \{\mathsf{v} \to \mathit{ProcDep}_1\} \end{array}\right] \right\rangle \right\}\right\}$$

$$\mathsf{false} \to \left\{\mathsf{state} \to \left\{\mathsf{procs} \to \left\langle \mathsf{Nothing} \triangleright \mathsf{j} : \left[\begin{array}{ll} \mathsf{None} & \to \mathsf{Everything} \\ \mathsf{Some} & \to \{\mathsf{v} \to \mathit{ProcDep}_2\} \end{array}\right] \right\rangle \right\}\right\}$$

where $\mathit{ProcDep}_1$ is:

$$\left\{\begin{array}{lll} & & \left[\begin{array}{lll} \mathsf{READ} & \to \{ & \mathsf{base} \to \mathsf{Everything}; \\ & & \mathsf{len} \to \mathsf{Everything}\} \\ \mathsf{WRITE} & \to \{ & \mathsf{base} \to \mathsf{Everything}; \\ & & \mathsf{len} \to \mathsf{Everything}\} \\ \mathsf{NONE} & \to \mathsf{Impossible} & \end{array}\right] \\ \mathsf{mem\_auth} & \to & \\ \mathsf{stackframe} & \to \{\mathsf{ds} \to \mathsf{Everything}\} & \end{array}\right\}$$

and $\mathit{ProcDep}_2$ is:

$$\left\{\begin{array}{lll} & & \left[\begin{array}{lll} \mathsf{READ} & \to \{ & \mathsf{base} \to \mathsf{Everything}; \\ & & \mathsf{len} \to \mathsf{Everything}\} \\ \mathsf{WRITE} & \to \{ & \mathsf{base} \to \mathsf{Everything}; \\ & & \mathsf{len} \to \mathsf{Everything}\} \\ \mathsf{NONE} & \to \mathsf{Nothing} & \end{array}\right] \\ \mathsf{mem\_auth} & \to & \\ \mathsf{stackframe} & \to \{\mathsf{ds} \to \mathsf{Everything}\} & \end{array}\right\}$$

8.5 Decision Procedure Experiments

We have applied a basic prototype of the decision procedure using the dependency and correlation summaries computed for the RSM/FSP layers of ProvenCore.

Our prototype considers pairs of one logical property and one predicate. The logical property and the predicate must both operate on values of the same type. More precisely, one of the predicate's inputs, as well as one of its outputs and one of the logical property's inputs must all be of the same type. Our prototype attempts to

detect whether the logical property is preserved after the execution of the predicate. If several inputs or outputs are of the same type, all combinations are considered. Most implicit types were not considered when searching for property/predicate pairs, as they are less likely to yield successful results. For example, arguments of a primitive type like `int` are unlikely to be unaffected by the execution of the predicate.

This prototype automatically inspected all such property/predicate pairs found in the RSM/FSP layers. A property was considered to be preserved if its dependency summary for the argument involved, when translated to a set of equalities, formed a subset of the equalities implied by the predicate's correlation summary. Both the `true` and the `false` exit labels were considered independently, and the property is considered to be preserved (subject to some conditions) when it is preserved for either or both exit labels. More precisely, given a property $\pi(\bar{\imath})[true|false]$ and a predicate $p(\bar{\imath}')[\ell : \bar{o}']$, we report success when it can satisfy the following:

$$\exists\, i \in \bar{\imath}, i' \in \bar{\imath}', o' \in \bar{o}' \text{ such that } \Gamma(i) = \Gamma(i') = \Gamma(o') \tag{8.1}$$
$$\wedge\, \exists\, \ell \in \{true, false\} \tag{8.2}$$
$$\wedge\, E(j) \neq E(k) \wedge E'(j) \neq E'(k) \quad \forall j, k \in \{\bar{\imath}, \bar{\imath}', \bar{o}'\} \tag{8.3}$$
$$\text{when } j \text{ and } k \text{ are used as array indices} \tag{8.4}$$
$$\wedge\, \langle E, [\ Prop(\bar{\imath}[i \to i'])[true|false]\ ]\rangle \xrightarrow{\ell} E \tag{8.5}$$
$$\wedge\, \langle E, [\ Pred(\bar{\imath}')[\ell' : \bar{o}|\ldots]\ ]\rangle \xrightarrow{\ell'} E' \tag{8.6}$$
$$\wedge\, \langle E' [\ Prop(\bar{\imath}[i \to o'])[true|false]\ ]\rangle \xrightarrow{\ell} E' \tag{8.7}$$

where $\bar{\imath}[i \to i']$ and $\bar{\imath}[i \to o']$ denote the sequence of variables $\bar{\imath}$ in which the variable i is replaced by the variable i' (respectively o').

This initial prototype was run on the 398 explicit predicates and 235 properties of the RSM/FSP layer of ProvenCore. Out of these, we filtered predicate/property pairs for which the property has an input i of the same type as one of the predicate's inputs i' and one of its outputs o'. These pairs involve 161 distinct predicates and 165 distinct properties. In total, there were 8250 tuples (i, i', o', ℓ) which satisfied the conditions 8.1 and 8.2.

This experiment allowed us, as a first result, to automatically identify 102 predicates for which at least one property is preserved under the conditions 8.1 – 8.7 stated above. For many predicates, it was possible to show that, after the execution of said predicate, several properties are preserved (up to 33). Figure 8.2 shows an overview of how many properties were inferred to be preserved for each predicate. The blue region at the bottom indicates how many properties are inferred to be preserved for a given predicate, while the red region above shows how many properties were compatible with the predicate, but were not inferred to be preserved.

Figure 8.3 shows an overview of how many predicates were inferred to be preserving each property. The blue region at the bottom indicates how many predicates are inferred to be preserving a given property, while the red region above shows how many

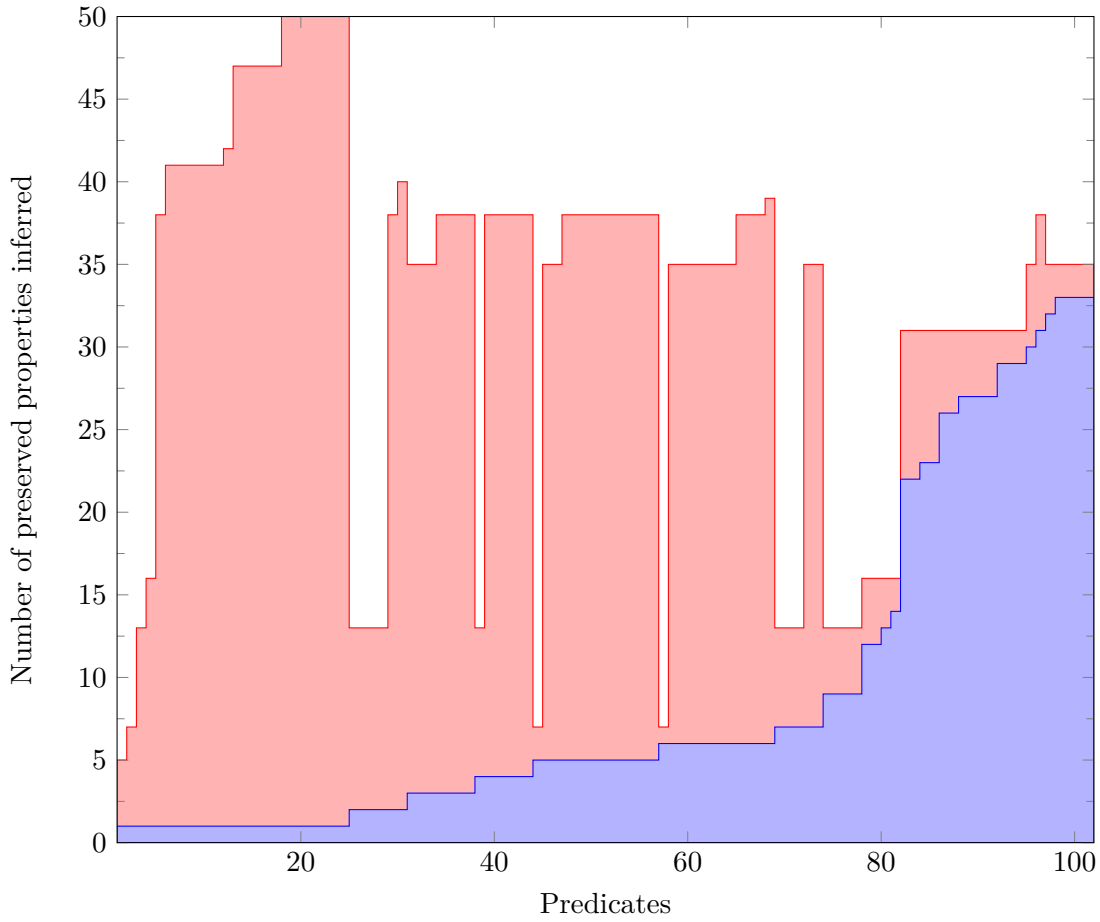


FIGURE 8.2 – Distribution of the number of inferred preserved properties. Predicates are sorted along that criterion.

predicates were compatible with the property, but were not inferred to be preserving it.

It is worth noting that in both figures 8.2 and 8.3, the red zone contains properties (respectively predicates) which could fall into these cases:

- The property is actually threatened by the predicate (respectively the predicate threatens the property);
- The property is not threatened (respectively the predicate is not threatening), but proving so requires more information that is obtained by our dependency and correlation analysis. For example, a more precise dependency or correlation analysis (e.g. tracking constructor evolution as presented in 7.6) could be needed. A numerical or value analysis could also help determine that the parts of the input data structure which are modified by the predicate and on which the logical

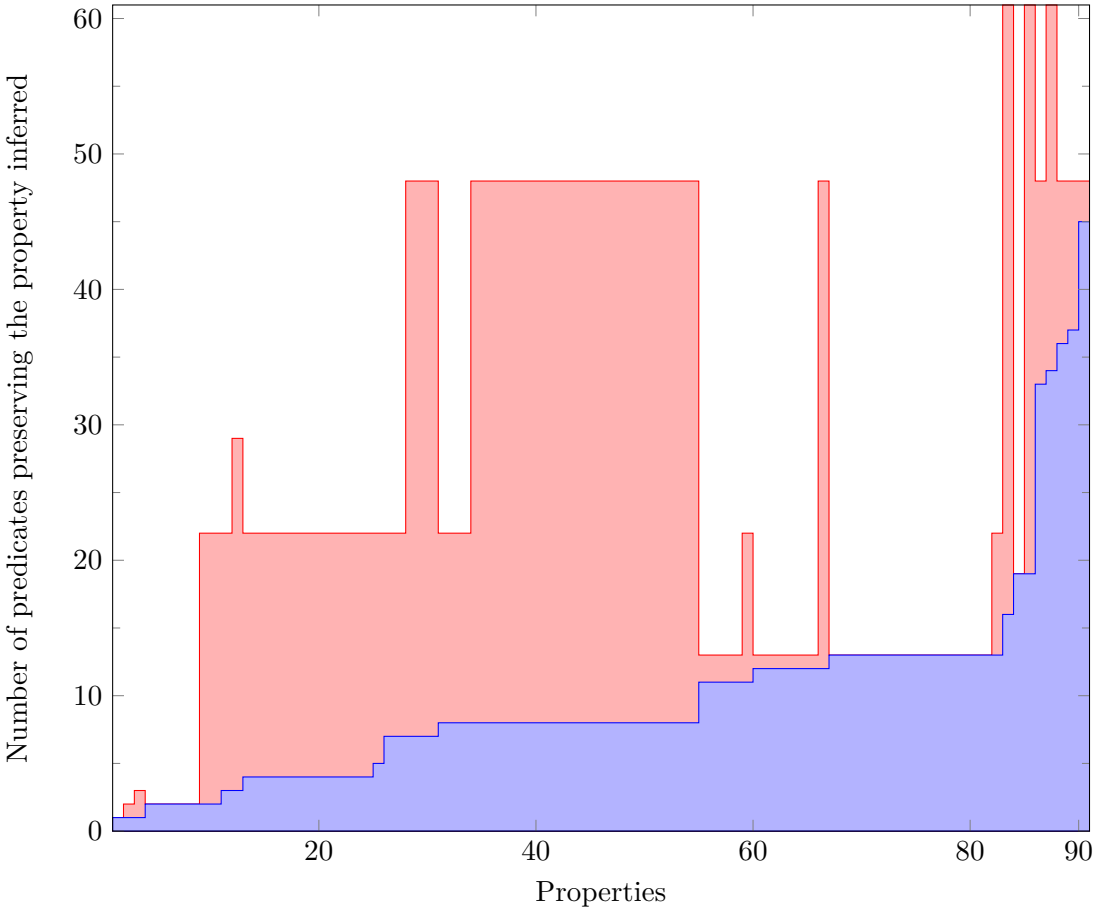


FIGURE 8.3 – Distribution of the number of inferred predicates for which a property is preserved. Properties are sorted along that criterion.

property also depends still satisfy the property after the execution of the predicate. Alternatively, the preservation of these properties can be demonstrated using an interactive prover.

- The property is not threatened (respectively the predicate is not threatening), and the dependency and correlation summaries contain enough information to prove the non-interference of the predicate and property, but our decision procedure prototype failed to infer it. This can be due to a timeout (this initial prototype has not been optimized at all, and can take a substantial time in some cases), or to precision losses in the decision procedure prototype itself.

Chapter 9

Conclusion and Perspectives

> There is no real ending. It's just the place where you stop the story.
>
> Frank Herbert

Despite its intuitive simplicity, the frame problem has proved to be an enduring issue with notoriously tedious implications. Its different manifestations have been studied for several decades in various contexts, ranging from Artificial Intelligence, in the context of which it has been originally identified, to the field of formal specification and verification. Recently, it has received extensive attention from the object-oriented verification community where it has been identified as a subsisting problem (Leavens, Leino, and Müller, 2007) and an ideal candidate for automation (Meyer, 2015). Classical approaches to addressing the frame problem are typically relying on separation logic (Reynolds, 2005) or ownership types (Clarke, Potter, and Noble, 1998). Though the merits of such approaches are indisputable, the manual specification effort that they require is non-negligible as well. Frame properties are an integral part of a complete specification and they are mandatory for proving correctness, but ideally, they should impose little additional effort. Programmers should be able to focus on the truly interesting part, namely what code does, and rely on automatic tools for the repetitive and cumbersome task of specifying and verifying frame properties.

Interactive formal verification of complex transition systems is not exempt from the manifestations of the frame problem either. Considerable effort is spent on proving the preservation of the system's invariants, even though in practice the majority of operations have a localised effect on the system and impact only a limited number of invariants at the same time. Identifying those invariants that are unaffected by an operation and automatically proving their preservation can substantially ease the proof burden for the programmer. In this thesis we have presented an approach towards automatically inferring the preservation of framing-related invariants. It is meant to be used in the context of an interactive theorem prover and employs two different static analyses, namely a dependency analysis and a correlation analysis, whose unified results are meant to establish the disjointness between the data dependencies of a logical property and the modifications performed by an operation. The decision procedure meant to combine the results of the two analyses is still in an incipient stage. However, our preliminary experiments related to automatically answering queries regarding the

preservation of certain invariants for unmodified parts are encouraging. We believe that our envisioned approach can become applicable to complex transition systems on a routine basis. Reasoning about framing can come for free without imposing the specification of additional clauses. We also believe that automatic reasoning about framing can be achieved through static analysis. Generally, static analysis has been considered prohibitive in terms of execution time. It has been predominantly used in an automatic context and avoided in interactive contexts where queries have to be answered fast so as not to impede the natural flow of an interactive proof. Though currently applied only on medium-sized models, given the short execution times of our dedicated static analyses, we believe that reasonable execution times for larger models can be expected as well. Therefore, we surmise that static analysis is applicable in an interactive verification context.

9.1 Contributions

The main contributions of this thesis are the designed and implemented dependency and correlation analyses, which are meant to be used in the context of an interactive theorem prover. Both analyses handle associative arrays and algebraic data types and compute fine-grained results mirroring the layered structures of such types. They target complex transition systems in general, and operating systems in particular. These are characterized by states defined by complex compound data structures and by transitions, i.e. state changes, that map an input state to an output state. Both of our static analyses are concerned with deep-state manipulations, i.e. accesses and modifications, respectively.

The dependency analysis presented in Chapter 5 automatically detects the relevant input subset needed for producing certain outputs. It handles functions and their specifications in a unified manner and computes for each possible execution scenario a conservative approximation of the input (sub)elements on which their outcome depends. It is a flow-sensitive, path-sensitive interprocedural data-flow analysis. Furthermore, for variants, an additional analysis is simultaneously conducted for computing the subset of possible constructors on a given execution scenario. Together with the dependency information per se, this additional information about constructors is meant to answer the same question, namely, what fragments of the input influence the output, from a different, albeit related point of view. The first version of the dependency analysis was fully context-insensitive. In order to introduce a relaxed form of context-sensitivity we have devised an extension based on symbolic paths. This was presented in Chapter 6.

The extension for the dependency analysis is based on computing deferred dependencies consisting of symbolic access maps in which callers can subsequently inject their specific context information on an as-needed basis. The dependency summaries for each predicate are still computed only once. However, by including nested context-sensitive components at the summaries' leaves, we reduce the precision penalty exerted by the fully context-insensitive approach without sacrificing performance. As discussed in Chapter 8, the deferred dependencies extension led to an increase of 10%–20% in

execution time on the used benchmarks. In terms of precision, it led to more precise dependency summaries for 50% of the predicates of the same benchmarks.

We surmise that besides its intended target, other programming activities can rely on our dependency analysis as well. For instance, the analysis can have applications in the testing realm, for designing and generating test suites that avoid redundant testing of the same execution scenario. Classes of inputs that will test the same execution scenario can be automatically determined. The input subelements on which the outputs of a predicate do not depend can be consistently supplied with the same testing value, as they are completely irrelevant for the outcome. On the contrary, the input subelements on which the outputs depend, should be targeted and their values should be varied for more comprehensive testing. Furthermore, our dependency analysis could also facilitate unit testing for exceptions as it computes specific results for every execution scenario of a predicate. Indeed, it is useful to have dedicated test cases which trigger each exception that can be thrown by a function. The set of relevant parts of the input differs for each possible exception and for the regular execution behaviour.

Our second contribution is the correlation analysis presented in Chapter 7 which detects the flow of input values into output values. It computes a conservative approximation of fine-grained equivalences between the input and the output subelements of a function. The correlation analysis is an interprocedural data-flow analysis that tracks the origin of subparts of the output and relates it to subparts of the inputs, thus summarising the behaviour of functions and detecting not only what is modified, but also how and to what extent. We have defined a partial equivalence type mirroring the layered structure of algebraic data types and associative arrays and we introduced an intermediate level consisting of access paths and correlations. These allow computing expressive information regarding equivalences between subparts of the inputs and subparts of the outputs in a flexible manner.

Prototypes for both of our analyses have been implemented in OCaml. These were discussed in Chapter 8. We have applied them to a functional specification of *ProvenCore* (Lescuyer, 2015), a general-purpose microkernel that ensures isolation. Results for medium-sized models have been obtained on average in less than 1 second with the dependency analysis, and less than 0.5 seconds on average with the correlation analysis. Static approaches have long been considered as being confined to small programs. We believe that our preliminary results indicate that it is possible to report conservative, precise information without sacrificing scalability.

We remark that our experience with the design and implementation of the two analyses has been rather different. The dependency analysis is much more complex semantically. This is partly a consequence of the simultaneous possible-constructors analysis, which has an impact on the abstract dependency domain. Deferred dependencies add yet another layer of complexity. However, the implementation proved to be much simpler than the implementation of the correlation analysis. The latter posed challenges due to the intermediate layer of access paths and correlations that we had to add for obtaining expressive, fine-grained information. However, the correlation analysis is simpler from a semantics point of view. It is also noteworthy to remark that for both analyses, an intermediate level below variables needed to be introduced, as soon as

fine-grained relations between pairs of variables were considered, directly or indirectly. In the case of deferred dependencies this was not the main goal, but rather a mechanism for obtaining increased precision in specific cases for already pertinent dependency information. In contrast, for the correlation analysis, the inclusion of an intermediate level was imperative for obtaining useful, expressive information in non-trivial cases.

As a first step towards a solution for automatically inferring the preservation of framing-related invariants, we have sketched a decision procedure meant to employ our two static analyses. By uncovering equivalences between inputs and outputs, after having detected that a property only depends on unmodified parts and by unifying the results, the preservation of invariants for the unmodified parts can be inferred.

9.2 Future Work

We conclude this thesis with some perspectives for practical future work, as well as some theoretical open issues that we wish to address in the future.

Practical Future Work. From a practical point of view, our future work goals revolve around the full implementation of the decision procedure, its integration in the interactive theorem prover developed at Prove & Run, as well as its comprehensive assessment in a real-word context.

Decision Procedure Implementation. Our first and main goal for the near future focuses on the full implementation of the decision procedure combining our dependency and correlation summaries and answering queries related to the preservation of logical properties. The performance of the algorithm sketched in Section 8.4 should be assessed on real-world examples. The complexity of this algorithm depends on the number of paths relating two endpoints in the graph of query atoms variables. It also depends on the number of correlations relating pairs of variables along the chains connecting endpoints. This could lead to a combinatorial explosion of the number of *compose* operations for large query graphs. Further optimization manners should be investigated and applied in the algorithm implementing the decision procedure.

Validation. After having implemented the decision procedure, the precision of our two static analyses employed by it should be comprehensively assessed on various benchmarks.

Some of the theoretical aspects related to our static analyses have been formalized in Coq by Stéphane Lescuyer. However, the actual implementation of the algorithms is not formally connected to the mechanized proofs. Therefore, it would be desirable to extensively test the implementation of the analysis algorithms. This could be done by translating the dependencies and correlations to types in a sufficiently expressive type system or by inserting runtime guards. These guards would check equalities for correlations and would taint supposedly irrelevant values identified by the dependency analysis, verifying that the output is not tainted. For the correlation analysis, inputs

which are correlated to some output values could be given a universally quantified type, the same type appearing in the parts of the output which are supposed to be equal. This is commonly used as a design pattern in functional programming languages to express data-flow constraints via the type system. For the dependency analysis, each part of the input which is supposed to be irrelevant for a predicate's output could be assigned a distinct polymorphic type variable which does not appear in the output. This allows the body of the predicate to take notice of a value's presence without being able to manipulate its contents.

Tool Integration and Support. Another important goal for the near future is the integration of our decision procedure in the ProvenTools interactive prover. A tactic allowing to automate the inference of framing-related invariant preservation should be supported. This goal entails a sequence of other considerations that have to be addressed. Currently, the dependency and correlation analyses handle whole programs and compute summaries for every predicate of the analysed program. Though the execution times of our analyses are low, even these can prove to be cumbersome in a real world context. Therefore, the two analyses should be adapted so as to allow incrementally analysing only parts of a program. Caching the results of the analyses across invocations of the decision procedure could prove to be efficient as well. Additionally, the mechanism of answering queries regarding invariant preservation should be transparent, allowing users to see the reasoning steps behind the decision procedure. Transparency is necessary for the ProvenTools prover which targets products that have to be certified. This possibly also requires a more concise output notation for the dependency and correlation summaries in order to ease the interpretation of results. Currently, they tend to be rather verbose for predicates handling composite values with a large number of subelements.

For the dependency summaries, a parser was implemented allowing users to annotate predicates with expected dependency information. A similar parser could be written for the correlation summaries. These annotations are a useful tool for testing the analyses on benchmarks for which the correlations and dependencies are known. In addition, they would allow users to annotate programs with constraints on the expected dependencies and correlations, similarly to type annotations in the presence of type inference and check that these expectations hold.

Finally, the decision procedure and our dependency and correlation analyses could be offered as a software library. A public API should describe and prescribe the expected behavior of our two static analyses and the decision procedure relying on them.

Theoretical Perspective. From a theoretical perspective, several interesting aspects remain open. In a nutshell, these consist in developing support for more sophisticated queries that could be answered by our decision procedure. The precision of our dependency and correlation analysis can be further increased as well.

Decision Procedure. A first interesting theoretical effort revolves around the formalization of our envisioned decision procedure used for inferring framing-related invariants. The types of queries it can answer should be further investigated and extended. For instance, it would be desirable to assert as a hypothesis that certain predicates are known to be valid on some nodes of the graph. We further identified two extensions for our correlation analysis that could increase the number of answered queries.

Constructor Evolution. For increasing the number of queries that our decision procedure can answer, one direction to investigate is the extension of our correlation analysis, in order to track and compute information regarding the evolution of variant constructors. This additional information should be leveraged to the context of our decision procedure. The formalization and implementation of this extension constitute an interesting effort. Furthermore, other types of relations between variables could be considered as well.

Correlations between Inputs. Another extension of our correlation analysis that would enrich the types of queries that can be answered by our decision procedure consists in tracking correlations between pairs of inputs, in addition to the ones computed between pairs of inputs and outputs. Besides the unified treatment of both actual code and logical properties on the correlation analysis side, this would allow answering queries that consist in a single logical property on multiple input values that are additionally related by other facts. It would also allow detecting aliasing between variables used as array indices.

Numerical Analysis for Arrays. Arrays are a source of precision loss in both of our static analyses. Hence, it would be interesting to investigate the impact of using simple numerical abstractions (congruence modulo and linear abstract domains). The numerical analysis could otherwise be offloaded to an external SMT solver, such as Z3 or Alt-Ergo for instance. Symbolic evaluation of the arithmetic computations should also be possible. This would avoid precision losses when joining two dependencies or correlations with exceptional information on distinct index variables which prove to have the same integer value in practice. Eliminating this source of imprecision would likely benefit the analysis of loops over arrays.

In conclusion, we have devised and implemented two static analyses detecting the data dependencies of a logical property, as well as correlations between the inputs and the outputs of operations. Our first results on a functional model of a microkernel are encouraging, both in terms of precision and speed, making these analyses suitable to use in the context of interactive provers. Aside from incremental improvements on the precision of our analyses, the next steps are to combine them in order to detect invariants which are not affected by the execution of a predicate, and to integrate this

as a tactic in the ProvenTools theorem prover. We believe that reasoning about framing can come for free, without imposing additional annotations. Inferring the preservation of framing-related invariants through static analysis can become applicable on a routine basis for complex transition systems.

Bibliography

Abrial, Jean-Raymond, Stephen A. Schuman, and Bertrand Meyer (1980). "Specification Language". In: *On the Construction of Programs*, pp. 343–410.

Alpuente, María, Santiago Escobar, and Salvador Lucas (2007). "Removing Redundant Arguments Automatically". In: *TPLP* 7.1-2, pp. 3–35. URL: http://dx.doi.org/10.1017/S1471068406002869.

Andreescu, Oana F., Thomas Jensen, and Stéphane Lescuyer (2015). "Dependency Analysis of Functional Specifications with Algebraic Data Structures". In: *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Proceedings*, pp. 116–133. DOI: 10.1007/978-3-319-25423-4_8. URL: http://dx.doi.org/10.1007/978-3-319-25423-4_8.

Andreescu, Oana Fabiana, Thomas Jensen, and Stéphane Lescuyer (2016). "Correlating Structured Inputs and Outputs in Functional Specifications". In: *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, pp. 85–103. DOI: 10.1007/978-3-319-41591-8_7. URL: http://dx.doi.org/10.1007/978-3-319-41591-8_7.

Asati, Rahul, Amitabha Sanyal, Amey Karkare, and Alan Mycroft (2014). "Liveness-Based Garbage Collection". In: *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pp. 85–106. DOI: 10.1007/978-3-642-54807-9_5. URL: http://dx.doi.org/10.1007/978-3-642-54807-9_5.

Baier, Christel and Joost-Pieter Katoen (2008). *Principles of Model Checking*. MIT Press. ISBN: 978-0-262-02649-9.

Banerjee, Anindya, Mike Barnett, and David A. Naumann (2008). "Boogie Meets Regions: A Verification Experience Report". In: *Verified Software: Theories, Tools, Experiments: Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*. Ed. by Natarajan Shankar and Jim Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 177–191. ISBN: 978-3-540-87873-5. DOI: 10.1007/978-3-540-87873-5_16. URL: http://dx.doi.org/10.1007/978-3-540-87873-5_16.

Banerjee, Anindya and David A. Naumann (2014). "A Logical Analysis of Framing for Specifications with Pure Method Calls". In: *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*, pp. 3–20. DOI: 10.1007/978-3-319-12154-3_1.

Banerjee, Anindya, David A. Naumann, and Stan Rosenberg (2008). "Regional Logic for Local Reasoning about Global Invariants". In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pp. 387–411. DOI: 10.1007/978-3-540-70592-5_17. URL: http://dx.doi.org/10.1007/978-3-540-70592-5_17.

— (2013). "Local Reasoning for Global Invariants, Part I: Region Logic". In: *J. ACM* 60.3, 18:1–18:56. DOI: 10.1145/2485982. URL: http://doi.acm.org/10.1145/2485982.

Barnes, J. and Praxis Critical Systems Limited (1997). *High Integrity Ada: The SPARK Approach*. Programming Languages. Addison-Wesley. ISBN: 9780201175172. URL: https://books.google.fr/books?id=YoBGAAAAYAAJ.

Barnett, Michael and David A. Naumann (2004). "Friends Need a Bit More: Maintaining Invariants Over Shared State". In: *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, pp. 54–84. DOI: 10.1007/978-3-540-27764-4_5. URL: http://dx.doi.org/10.1007/978-3-540-27764-4_5.

Barnett, Michael, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte (2004). "Verification of Object-Oriented Programs with Invariants". In: *Journal of Object Technology* 3.6, pp. 27–56. DOI: 10.5381/jot.2004.3.6.a2. URL: http://dx.doi.org/10.5381/jot.2004.3.6.a2.

Barnett, Michael, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino (2005a). "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*, pp. 364–387. DOI: 10.1007/11804192_17. URL: http://dx.doi.org/10.1007/11804192_17.

Barnett, Michael, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter (2005b). "The Spec# Programming System: Challenges and Directions". In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pp. 144–152. DOI: 10.1007/978-3-540-69149-5_16. URL: http://dx.doi.org/10.1007/978-3-540-69149-5_16.

Barnett, Mike, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter (2011). "Specification and Verification: The Spec# Experience". In: *Commun. ACM* 54.6, pp. 81–91. DOI: 10.1145/1953122.1953145. URL: http://doi.acm.org/10.1145/1953122.1953145.

Berdine, Josh, Cristiano Calcagno, and Peter W. O'Hearn (2005). "Smallfoot: Modular Automatic Assertion Checking with Separation Logic". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pp. 115–137. DOI: 10.1007/11804192_6. URL: http://dx.doi.org/10.1007/11804192_6.

— (2012). "Verification Condition Generation and Variable Conditions in Smallfoot". In: *CoRR* abs/1204.4804. URL: http://arxiv.org/abs/1204.4804.

Berdine, Josh, Byron Cook, and Samin Ishtiaq (2011). "SLAyer: Memory Safety for Systems-Level Code". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pp. 178–183. DOI: 10.1007/978-3-642-22110-1_15. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_15.

Berg, Joachim van den and Bart Jacobs (2001). "The LOOP Compiler for Java and JML". In: *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pp. 299–312. DOI: 10.1007/3-540-45319-9_21. URL: http://dx.doi.org/10.1007/3-540-45319-9_21.

Bertot, Yves and Pierre Castéran (2004). *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5. URL: http://dx.doi.org/10.1007/978-3-662-07964-5.

Bertrane, Julien, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival (2015). "Static Analysis and Verification of Aerospace Software by Abstract Interpretation". In: *Foundations and Trends in Programming Languages* 2.2-3, pp. 71–190. DOI: 10.1561/2500000002. URL: http://dx.doi.org/10.1561/2500000002.

Blanchet, Bruno, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival (2003). "A Static Analyzer for Large Safety-Critical Software". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pp. 196–207. DOI: 10.1145/781131.781153. URL: http://doi.acm.org/10.1145/781131.781153.

Bobot, François and Jean-Christophe Filliâtre (2012). "Separation Predicates: A Taste of Separation Logic in First-Order Logic". In: *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, pp. 167–181. DOI: 10.1007/978-3-642-34281-3_14. URL: http://dx.doi.org/10.1007/978-3-642-34281-3_14.

Borgida, Alexander, John Mylopoulos, and Raymond Reiter (1993). ""... And Nothing Else Changes": The Frame Problem in Procedure Specifications". In: *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993*. Pp. 303–314. URL: http://portal.acm.org/citation.cfm?id=257572.257636.

— (1995). "On the Frame Problem in Procedure Specifications". In: *IEEE Trans. Software Eng.* 21.10, pp. 785–798. DOI: 10.1109/32.469460. URL: http://dx.doi.org/10.1109/32.469460.

Bouissou, O., É. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, É. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin (2009).

"Space Software Validation using Abstract Interpretation". In: *Proc. of the International Space System Engineering Conference on Data Systems in Aerospace (DASIA 2009)*. Vol. SP-669. http://www-apr.lip6.fr/~mine/publi/article-bouissou-al-dasia09.pdf. Istambul, Turkey: ESA, p. 7. DOI: 1921532.1921553.

Burdy, Lilian, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll (2005). "An Overview of JML Tools and Applications". In: *STTT* 7.3, pp. 212–232. DOI: 10.1007/s10009-004-0167-4. URL: http://dx.doi.org/10.1007/s10009-004-0167-4.

Calcagno, Cristiano and Dino Distefano (2011). "Infer: An Automatic Program Verifier for Memory Safety of C Programs". In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pp. 459–465. DOI: 10.1007/978-3-642-20398-5_33. URL: http://dx.doi.org/10.1007/978-3-642-20398-5_33.

Calcagno, Cristiano, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang (2008). "Space Invading Systems Code". In: *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008, Revised Selected Papers*, pp. 1–3. DOI: 10.1007/978-3-642-00515-2_1. URL: http://dx.doi.org/10.1007/978-3-642-00515-2_1.

— (2009). "Compositional Shape Analysis by Means of Bi-Abduction". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pp. 289–300. DOI: 10.1145/1480881.1480917. URL: http://doi.acm.org/10.1145/1480881.1480917.

— (2011). "Compositional Shape Analysis by Means of Bi-Abduction". In: *J. ACM* 58.6, p. 26. DOI: 10.1145/2049697.2049700.

Cardelli, Luca and Peter Wegner (1985). "On Understanding Types, Data Abstraction, and Polymorphism". In: *ACM Comput. Surv.* 17.4, pp. 471–522. DOI: 10.1145/6041.6042. URL: http://doi.acm.org/10.1145/6041.6042.

Castillo, Rosa, Francisco Corbera, Angeles G. Navarro, Rafael Asenjo, and Emilio L. Zapata (2008). "Complete Def-Use Analysis in Recursive Programs with Dynamic Data Structures". In: *Euro-Par 2008 Workshops - Parallel Processing, VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers*, pp. 273–282. DOI: 10.1007/978-3-642-00955-6_32. URL: http://dx.doi.org/10.1007/978-3-642-00955-6_32.

Cataño, Néstor and Marieke Huisman (2003). "CHASE: A Static Checker for JML's Assignable Clause". In: *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*, pp. 26–40. DOI: 10.1007/3-540-36384-X_6. URL: http://dx.doi.org/10.1007/3-540-36384-X_6.

Chalin, Patrice, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll (2005). "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*,

pp. 342–363. DOI: 10.1007/11804192_16. URL: http://dx.doi.org/10.1007/11804192_16.

Chang, Bor-Yuh Evan and K. Rustan M. Leino (2005). "Abstract Interpretation with Alien Expressions and Heap Structures". In: *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Proceedings*, pp. 147–163. DOI: 10.1007/978-3-540-30579-8_11. URL: http://dx.doi.org/10.1007/978-3-540-30579-8_11.

Clarke, David G. and Sophia Drossopoulou (2002). "Ownership, Encapsulation and the Disjointness of Type and Effect". In: *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002.* Pp. 292–310. DOI: 10.1145/582419.582447. URL: http://doi.acm.org/10.1145/582419.582447.

Clarke, David G., John Potter, and James Noble (1998). "Ownership Types for Flexible Alias Protection". In: *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.* Pp. 48–64. DOI: 10.1145/286936.286947. URL: http://doi.acm.org/10.1145/286936.286947.

Clarke, Edmund M. and E. Allen Emerson (1981). "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*, pp. 52–71. DOI: 10.1007/BFb0025774. URL: http://dx.doi.org/10.1007/BFb0025774.

Cok, David R. (2005). "Reasoning with Specifications Containing Method Calls and Model Fields". In: *Journal of Object Technology* 4.8, pp. 77–103. DOI: 10.5381/jot.2005.4.8.a4. URL: http://dx.doi.org/10.5381/jot.2005.4.8.a4.

Cousot, P. and R. Cousot (1994). "Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages), invited paper". In: *Proceedings of the 1994 International Conference on Computer Languages*. Toulouse, France: IEEE Computer Society Press, Los Alamitos, California, pp. 95–112.

Cousot, Patrick (2001). "Abstract Interpretation Based Formal Methods and Future Challenges". In: *Informatics - 10 Years Back. 10 Years Ahead.* Pp. 138–156. DOI: 10.1007/3-540-44577-3_10. URL: http://dx.doi.org/10.1007/3-540-44577-3_10.

Cousot, Patrick and Radhia Cousot (1977). "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*, pp. 238–252. DOI: 10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973.

— (2010). "A Gentle Introduction to Formal Verification of Computer Systems by Abstract Interpretation". In: *Logics and Languages for Reliability and Security*, pp. 1–29. DOI: 10.3233/978-1-60750-100-8-1. URL: http://dx.doi.org/10.3233/978-1-60750-100-8-1.

Cousot, Patrick, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival (2005). "The ASTREÉ 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*, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3. URL: http://dx.doi.org/10.1007/978-3-540-31987-0_3.

Cousot, Patrick, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival (2007). "Varieties of Static Analyzers: A Comparison with ASTREE". In: *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, June 5-8, 2007, Shanghai, China*, pp. 3–20. DOI: 10.1109/TASE.2007.55. URL: http://dx.doi.org/10.1109/TASE.2007.55.

Cuoq, Pascal, Virgile Prevosto, and Boris Yakobowski. *Frama-C Value Analysis User Manual*. URL: http://frama-c.com/download/frama-c-value-analysis.pdf.

Cuoq, Pascal, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski (2012). "Frama-C - A Software Analysis Perspective". In: *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, pp. 233–247. DOI: 10.1007/978-3-642-33826-7_16. URL: http://dx.doi.org/10.1007/978-3-642-33826-7_16.

Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck (1989). "An Efficient Method of Computing Static Single Assignment Form". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pp. 25–35. DOI: 10.1145/75277.75280. URL: http://doi.acm.org/10.1145/75277.75280.

Darvas, Ádám and Peter Müller (2006). "Reasoning About Method Calls in Interface Specifications". In: *Journal of Object Technology* 5.5, pp. 59–85. DOI: 10.5381/jot.2006.5.5.a3. URL: http://dx.doi.org/10.5381/jot.2006.5.5.a3.

Delmas, David and Jean Souyris (2007). "Astrée: From Research to Industry". In: *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, pp. 437–451. DOI: 10.1007/978-3-540-74061-2_27. URL: http://dx.doi.org/10.1007/978-3-540-74061-2_27.

Dietl, Werner and Peter Müller (2005). "Universes: Lightweight Ownership for JML". In: *Journal of Object Technology* 4.8, pp. 5–32. DOI: 10.5381/jot.2005.4.8.a1. URL: http://dx.doi.org/10.5381/jot.2005.4.8.a1.

Dijkstra, Edsger W. (1976). *A Discipline of Programming*. Prentice-Hall.

Distefano, Dino, Peter W. O'Hearn, and Hongseok Yang (2006). "A Local Shape Analysis Based on Separation Logic". In: *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'06. Vienna, Austria: Springer-Verlag, pp. 287–302. ISBN: 3-540-33056-9, 978-3-540-33056-1.

Distefano, Dino and Matthew J. Parkinson (2008). "jStar: Towards Practical Verification for Java". In: *Proceedings of the 23rd Annual ACM SIGPLAN Conference*

on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA, pp. 213–226. DOI: 10.1145/1449764.1449782. URL: http://doi.acm.org/10.1145/1449764.1449782.

Drossopoulou, Sophia, Adrian Francalanza, Peter Müller, and Alexander J. Summers (2008). "A Unified Framework for Verification Techniques for Object Invariants". In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pp. 412–437. DOI: 10.1007/978-3-540-70592-5_18. URL: http://dx.doi.org/10.1007/978-3-540-70592-5_18.

Eclipse Java Development Tools (JDT). http://www.eclipse.org/jdt/. Accessed: 2016-09-11.

Feijs, L. M. G. Loe M. G. and H. B. M. Jonkers (1992). *Formal Specification and Design*. Cambridge tracts in theoretical computer science. Cambridge, New York: Cambridge University Press. ISBN: 0-521-43457-2. URL: http://opac.inria.fr/record=b1083844.

Flanagan, Cormac, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata (2002). "Extended Static Checking for Java". In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pp. 234–245. DOI: 10.1145/512529.512558. URL: http://doi.acm.org/10.1145/512529.512558.

Floyd, Robert W. (1967). "Assigning Meanings to Programs". In: *Mathematical Aspects of Computer Science*. Ed. by J. T. Schwartz. Vol. 19. Proceedings of Symposia in Applied Mathematics. Providence, Rhode Island: American Mathematical Society, pp. 19–32.

Gallier, Jean H. (1987). *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Wiley. ISBN: 978-0-471-61546-0.

Gharat, Pritam M., Uday P. Khedker, and Alan Mycroft (2016). "Flow- and Context-Sensitive Points-To Analysis Using Generalized Points-To Graphs". In: *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pp. 212–236. DOI: 10.1007/978-3-662-53413-7_11. URL: http://dx.doi.org/10.1007/978-3-662-53413-7_11.

Greenhouse, Aaron and John Boyland (1999). "An Object-Oriented Effects System". In: *ECOOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, pp. 205–229. DOI: 10.1007/3-540-48743-3_10. URL: http://dx.doi.org/10.1007/3-540-48743-3_10.

Gross, Thomas R. and Peter Steenkiste (1990). "Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler". In: *Softw., Pract. Exper.* 20.2, pp. 133–155. DOI: 10.1002/spe.4380200203. URL: http://dx.doi.org/10.1002/spe.4380200203.

Guttag, John V., James J. Horning, and Jeannette M. Wing (1985). "The Larch Family of Specification Languages". In: *IEEE Software* 2.5, pp. 24–36. DOI: 10.1109/MS.1985.231756. URL: http://dx.doi.org/10.1109/MS.1985.231756.

Guttag, John V., James J. Horning, Stephen J. Garland, Kevin D. Jones, A. Modet, and Jeannette M. Wing (1993a). *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer. ISBN: 978-1-4612-7636-4.

DOI: 10.1007/978-1-4612-2704-5. URL: http://dx.doi.org/10.1007/978-1-4612-2704-5.

Guttag, John V., James J. Horning, Stephen J. Garland, Kevin D. Jones, A. Modet, and Jeannette M. Wing (1993b). *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer. ISBN: 978-1-4612-7636-4. DOI: 10.1007/978-1-4612-2704-5. URL: http://dx.doi.org/10.1007/978-1-4612-2704-5.

Hammer, Christian and Gregor Snelting (2009). "Flow-Sensitive, Context-Sensitive, and Object-Sensitive Information Flow Control based on Program Dependence Graphs". In: *Int. J. Inf. Sec.* 8.6, pp. 399–422. DOI: 10.1007/s10207-009-0086-1. URL: http://dx.doi.org/10.1007/s10207-009-0086-1.

Hatcliff, John, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson (2012). "Behavioral Interface Specification Languages". In: *ACM Comput. Surv.* 44.3, p. 16. DOI: 10.1145/2187671.2187678. URL: http://doi.acm.org/10.1145/2187671.2187678.

Heintze, Nevin and Olivier Tardieu (2001). "Demand-Driven Pointer Analysis". In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: ACM, pp. 24–34. ISBN: 1-58113-414-2. DOI: 10.1145/378795.378802. URL: http://doi.acm.org/10.1145/378795.378802.

Hind, Michael (2001). "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*, pp. 54–61. DOI: 10.1145/379605.379665. URL: http://doi.acm.org/10.1145/379605.379665.

Hoare, C. A. R. (1969). "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10, pp. 576–580. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

— (1971). "Procedures and Parameters: An Axiomatic Approach". In: *Symposium on Semantics of Algorithmic Languages*, pp. 102–116. DOI: 10.1007/BFb0059696. URL: http://dx.doi.org/10.1007/BFb0059696.

Horwitz, Susan, Thomas W. Reps, and Shmuel Sagiv (1995). "Demand Interprocedural Dataflow Analysis". In: *SIGSOFT '95, Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, Washington, DC, USA, October 10-13, 1995*, pp. 104–115. DOI: 10.1145/222124.222146. URL: http://doi.acm.org/10.1145/222124.222146.

Hughes, J. (1987). "Backwards Analysis of Functional Programs". In: *IFIP Workshop on Partial Evaluation and Mixed Computation*. Ed. by Bjørner and Ershov.

Hur, Chung-Kil, Derek Dreyer, and Viktor Vafeiadis (2011). "Separation Logic in the Presence of Garbage Collection". In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pp. 247–256. DOI: 10.1109/LICS.2011.46. URL: http://dx.doi.org/10.1109/LICS.2011.46.

Jacobs, Bart and Frank Piessens (2006). "Verification of Programs with Inspector Methods". In: *In FTfJP 2006*.

Jacobs, Bart, Jan Smans, and Frank Piessens (2010). "A Quick Tour of the VeriFast Program Verifier". In: *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, pp. 304–311. DOI: 10.1007/978-3-642-17164-2_21. URL: http://dx.doi.org/10.1007/978-3-642-17164-2_21.

Jacobs, Bart, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens (2011). "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4. URL: http://dx.doi.org/10.1007/978-3-642-20398-5_4.

Java Native Interface Documentation (JNI). URL: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/intro.html#wp725 (Accessed: 09/11/2016).

Jensen, Simon Holm, Anders Møller, and Peter Thiemann (2010). "Interprocedural Analysis with Lazy Propagation". In: *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, pp. 320–339. DOI: 10.1007/978-3-642-15769-1_20. URL: http://dx.doi.org/10.1007/978-3-642-15769-1_20.

Jhala, Ranjit and Rupak Majumdar (2009). "Software Model Checking". In: *ACM Comput. Surv.* 41.4, 21:1–21:54. DOI: 10.1145/1592434.1592438. URL: http://doi.acm.org/10.1145/1592434.1592438.

Jones, Cliff B. (1990). *Systematic Software Development Using VDM (2Nd Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN: 0-13-880733-7.

Jones, Neil D. and Steven S. Muchnick (1979). "Flow Analysis and Optimization of Lisp-Like Structures". In: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, 1979*, pp. 244–256. DOI: 10.1145/567752.567776. URL: http://doi.acm.org/10.1145/567752.567776.

Jones, Simon B. and Daniel Le Métayer (1989). "Computer-Time Garbage Collection by Sharing Analysis". In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pp. 54–74. DOI: 10.1145/99370.99375. URL: http://doi.acm.org/10.1145/99370.99375.

Kassios, Ioannis T. (2006). "Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions". In: *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, pp. 268–283. DOI: 10.1007/11813040_19. URL: http://dx.doi.org/10.1007/11813040_19.

— (2011). "The Dynamic Frames Theory". In: *Formal Asp. Comput.* 23.3, pp. 267–288. DOI: 10.1007/s00165-010-0152-5. URL: http://dx.doi.org/10.1007/s00165-010-0152-5.

Kennedy, Ken (1978). "Use-Definition Chains with Applications". In: *Comput. Lang.* 3.3, pp. 163–179. DOI: 10.1016/0096-0551(78)90009-7. URL: http://dx.doi.org/10.1016/0096-0551(78)90009-7.

Khedker, Uday P., Alan Mycroft, and Prashant Singh Rawat (2011). "Lazy Pointer Analysis". In: *CoRR* abs/1112.5000. URL: http://arxiv.org/abs/1112.5000.

Kildall, Gary A. (1973). "A Unified Approach to Global Program Optimization". In: *Conference Record of the ACM Symposium on Principles of Programming Languages, 1973*, pp. 194–206. DOI: 10.1145/512927.512945. URL: http://doi.acm.org/10.1145/512927.512945.

Klein, Gerwin, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood (2009). "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://doi.acm.org/10.1145/1629575.1629596.

Knoop, Jens, Oliver Rüthing, and Bernhard Steffen (1994). "Partial Dead Code Elimination". In: *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pp. 147–158. DOI: 10.1145/178243.178256. URL: http://doi.acm.org/10.1145/178243.178256.

Koenig, Jason and K. Rustan M. Leino (2012). "Getting Started with Dafny: A Guide". In: *Software Safety and Security - Tools for Analysis and Verification*, pp. 152–181. DOI: 10.3233/978-1-61499-028-4-152. URL: http://dx.doi.org/10.3233/978-1-61499-028-4-152.

Kogtenkov, Alexander, Bertrand Meyer, and Sergey Velder (2015). "Alias Calculus, Change Calculus and Frame Inference". In: *Sci. Comput. Program.* 97.P1, pp. 163–172. ISSN: 0167-6423.

Lattner, Chris, Andrew Lenharth, and Vikram S. Adve (2007). "Making Context-Sensitive Points-To Analysis with Heap Cloning Practical for the Real World". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, 2007*, pp. 278–289. DOI: 10.1145/1250734.1250766. URL: http://doi.acm.org/10.1145/1250734.1250766.

Leavens, Gary T., Albert L. Baker, and Clyde Ruby (2006). "Preliminary Design of JML: A Behavioral Interface Specification Language for Java". In: *ACM SIGSOFT Software Engineering Notes* 31.3, pp. 1–38. DOI: 10.1145/1127878.1127884. URL: http://doi.acm.org/10.1145/1127878.1127884.

Leavens, Gary T. and Curtis Clifton (2005). "Lessons from the JML Project". In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pp. 134–143. DOI: 10.1007/978-3-540-69149-5_15. URL: http://dx.doi.org/10.1007/978-3-540-69149-5_15.

Leavens, Gary T., K. Rustan M. Leino, and Peter Müller (2007). "Specification and Verification Challenges for Sequential Object-Oriented Programs". In: *Formal Asp.*

Comput. 19.2, pp. 159–189. DOI: 10.1007/s00165-007-0026-7. URL: http://dx.doi.org/10.1007/s00165-007-0026-7.

Leavens, Gary T. and Peter Müller (2007). "Information Hiding and Visibility in Interface Specifications". In: *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pp. 385–395. DOI: 10.1109/ICSE.2007.44. URL: http://dx.doi.org/10.1109/ICSE.2007.44.

Leavens, Gary T., Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry (2006). *JML Reference Manual*.

Lehner, Hermann and Peter Müller (2010). "Efficient Runtime Assertion Checking of Assignable Clauses with Datagroups". In: *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pp. 338–352. DOI: 10.1007/978-3-642-12029-9_24. URL: http://dx.doi.org/10.1007/978-3-642-12029-9_24.

Leinenbach, Dirk and Thomas Santen (2009). "Verifying the Microsoft Hyper-V Hypervisor with VCC". In: *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*. Ed. by Ana Cavalcanti and Dennis R. Dams. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 806–809. ISBN: 978-3-642-05089-3. DOI: 10.1007/978-3-642-05089-3_51. URL: http://dx.doi.org/10.1007/978-3-642-05089-3_51.

Leino, K. Rustan M. *This is Boogie 2, Boogie Reference Manual*. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf.

— (1998). "Data Groups: Specifying the Modification of Extended State". In: *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998*. Pp. 144–153. DOI: 10.1145/286936.286953. URL: http://doi.acm.org/10.1145/286936.286953.

— (2001). "Extended Static Checking: A Ten-Year Perspective". In: *Informatics - 10 Years Back. 10 Years Ahead.* Pp. 157–175. DOI: 10.1007/3-540-44577-3_11. URL: http://dx.doi.org/10.1007/3-540-44577-3_11.

— (2010). "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20. URL: http://dx.doi.org/10.1007/978-3-642-17511-4_20.

Leino, K. Rustan M. and Peter Müller (2004). "Object Invariants in Dynamic Contexts". In: *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, pp. 491–516. DOI: 10.1007/978-3-540-24851-4_22. URL: http://dx.doi.org/10.1007/978-3-540-24851-4_22.

— (2006). "A Verification Methodology for Model Fields". In: *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*

2006, Vienna, Austria, March 27-28, 2006, Proceedings, pp. 115–130. DOI: 10.1007/11693024_9. URL: http://dx.doi.org/10.1007/11693024_9.

Leino, K. Rustan M. and Peter Müller (2008a). "Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs". In: *Advanced Lectures on Software Engineering, LASER Summer School 2007/2008*, pp. 91–139. DOI: 10.1007/978-3-642-13010-6_4. URL: http://dx.doi.org/10.1007/978-3-642-13010-6_4.

— (2008b). "Verification of Equivalent-Results Methods". In: *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pp. 307–321. DOI: 10.1007/978-3-540-78739-6_24. URL: http://dx.doi.org/10.1007/978-3-540-78739-6_24.

Leino, K. Rustan M., Peter Müller, and Jan Smans (2009). "Verification of Concurrent Programs with Chalice". In: *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, pp. 195–222. DOI: 10.1007/978-3-642-03829-7_7. URL: http://dx.doi.org/10.1007/978-3-642-03829-7_7.

Leino, K. Rustan M., Peter Müller, and Angela Wallenburg (2008). "Flexible Immutability with Frozen Objects". In: *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, pp. 192–208. DOI: 10.1007/978-3-540-87873-5_17. URL: http://dx.doi.org/10.1007/978-3-540-87873-5_17.

Leino, K. Rustan M. and Greg Nelson (1998). "An Extended Static Checker for Modular-3". In: *Compiler Construction, 7th International Conference, CC'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, pp. 302–305. DOI: 10.1007/BFb0026441. URL: http://dx.doi.org/10.1007/BFb0026441.

— (2002). "Data Abstraction and Information Hiding". In: *ACM Trans. Program. Lang. Syst.* 24.5, pp. 491–553. DOI: 10.1145/570886.570888. URL: http://doi.acm.org/10.1145/570886.570888.

Leino, K. Rustan M., Arnd Poetzsch-Heffter, and Yunhong Zhou (2002). "Using Data Groups to Specify and Check Side Effects". In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pp. 246–257. DOI: 10.1145/512529.512559. URL: http://doi.acm.org/10.1145/512529.512559.

Leino, K. Rustan M. and Philipp Rümmer (2010). "A Polymorphic Intermediate Verification Language: Design and Logical Encoding". In: *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pp. 312–327. DOI: 10.1007/978-3-642-12002-2_26. URL: http://dx.doi.org/10.1007/978-3-642-12002-2_26.

Leroy, Xavier (2009). "A Formally Verified Compiler Back-end". In: *J. Autom. Reasoning* 43.4, pp. 363–446. DOI: 10.1007/s10817-009-9155-4. URL: http://dx.doi.org/10.1007/s10817-009-9155-4.

Leroy, Xavier and François Pessaux (2000). "Type-Based Analysis of Uncaught Exceptions". In: *ACM Trans. Program. Lang. Syst.* 22.2, pp. 340–377. DOI: 10.1145/349214.349230. URL: http://doi.acm.org/10.1145/349214.349230.

Lescuyer, Stéphane (2015). "ProvenCore: Towards a Verified Isolation Micro-Kernel". In: International Workshop on MILS: Architecture and Assurance for Secure Systems. URL: http://mils-workshop-2015.euromils.eu/.

Leuschel, Michael and Morten Heine Sørensen (1996). "Redundant Argument Filtering of Logic Programs". In: *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, pp. 83–103. DOI: 10.1007/3-540-62718-9_6. URL: http://dx.doi.org/10.1007/3-540-62718-9_6.

Lhoták, Ondrej and Laurie J. Hendren (2006). "Context-Sensitive Points-to Analysis: Is It Worth It?" In: *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*, pp. 47–64. DOI: 10.1007/11688839_5. URL: http://dx.doi.org/10.1007/11688839_5.

Liang, Sheng (1999). *Java Native Interface: Programmer's Guide and Reference*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201325772.

Liskov, Barbara and John Guttag (1986). *Abstraction and Specification in Program Development*. Cambridge, MA, USA: MIT Press. ISBN: 0-262-12112-3.

Liu, Yanhong A. (1998). "Dependence Analysis for Recursive Data". In: *Proceedings of the 1998 International Conference on Computer Languages, ICCL 1998, Chicago, IL, USA, May 14-16, 1998*, pp. 206–215. DOI: 10.1109/ICCL.1998.674171. URL: http://dx.doi.org/10.1109/ICCL.1998.674171.

Liu, Yanhong A. and Scott D. Stoller (2003). "Eliminating Dead Code on Recursive Data". In: *Sci. Comput. Program.* 47.2-3, pp. 221–242. DOI: 10.1016/S0167-6423(02)00134-X. URL: http://dx.doi.org/10.1016/S0167-6423(02)00134-X.

Lu, Yi, John Potter, and Jingling Xue (2007). "Validity Invariants and Effects". In: *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, pp. 202–226. DOI: 10.1007/978-3-540-73589-2_11. URL: http://dx.doi.org/10.1007/978-3-540-73589-2_11.

Marché, Claude, Christine Paulin-Mohring, and Xavier Urbain (2004). "The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs Annotated in JML". In: *J. Log. Algebr. Program.* 58.1-2, pp. 89–106. DOI: 10.1016/j.jlap.2003.07.006. URL: http://dx.doi.org/10.1016/j.jlap.2003.07.006.

Marché, Claude (2016). *The Krakatoa Verification Tool for Java Programs, Krakatoa Tutorial and Reference Manual*. URL: http://krakatoa.lri.fr/krakatoa.pdf.

Martin-Löf, Per (1984). *Intuitionistic Type Theory*. Naples: Bibliopolis.

McCarthy, John and Patrick J. Hayes (1969). "Some Philosophical Problems from the Standpoint of Artificial Intelligence". In: *Machine Intelligence*. Edinburgh University Press.

Meyer, Bertrand (1991). *Eiffel: The Language*. Prentice-Hall. ISBN: 0-13-247925-7.

— (1992). "Applying "Design by Contract"". In: *IEEE Computer* 25.10, pp. 40–51. DOI: 10.1109/2.161279. URL: http://dx.doi.org/10.1109/2.161279.

Meyer, Bertrand (1997). *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall. ISBN: 0-13-629155-4.

— (2010). "Towards a Theory and Calculus of Aliasing". In: *Journal of Object Technology* 9.2, pp. 37–74. DOI: 10.5381/jot.2010.9.2.c5. URL: http://dx.doi.org/10.5381/jot.2010.9.2.c5.

— (2011). "Steps Towards a Theory and Calculus of Aliasing". In: *Int. J. Software and Informatics* 5.1-2, pp. 77–115. URL: http://www.ijsi.org/ch/reader/view_abstract.aspx?file_no=i77.

— (2015). "Framing the Frame Problem". In: *Dependable Software Systems Engineering*, pp. 193–203. DOI: 10.3233/978-1-61499-495-4-193. URL: http://dx.doi.org/10.3233/978-1-61499-495-4-193.

Midtgaard, Jan (2012). "Control-Flow Analysis of Functional Programs". In: *ACM Comput. Surv.* 44.3, p. 10. DOI: 10.1145/2187671.2187672. URL: http://doi.acm.org/10.1145/2187671.2187672.

Mike Barnett Rustan Leino, Wolfram Schulte (2005). "The Spec# Programming System: An Overview". In: *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*. Vol. 3362. Springer, pp. 49–69. URL: https://www.microsoft.com/en-us/research/publication/the-spec-programming-system-an-overview/.

Milanova, Ana, Atanas Rountev, and Barbara G. Ryder (2005). "Parameterized Object Sensitivity for Points-To Analysis for Java". In: *ACM Trans. Softw. Eng. Methodol.* 14.1, pp. 1–41. DOI: 10.1145/1044834.1044835. URL: http://doi.acm.org/10.1145/1044834.1044835.

Montenegro, Manuel, Ricardo Peña, and Clara Segura (2015). "Shape Analysis in a Functional Language by Using Regular Languages". In: *Sci. Comput. Program.* 111, pp. 51–78. DOI: 10.1016/j.scico.2014.12.006. URL: http://dx.doi.org/10.1016/j.scico.2014.12.006.

Morgenstern, Leora (1995). "The Problem with Solutions to the Frame Problem". In: *The Robot's Dilemma Revisited: The Frame Problem in Artificial Intelligence. Ablex*. Ablex Publishing Co, pp. 99–133.

Moura, Leonardo Mendonça de and Nikolaj Bjørner (2008). "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: http://dx.doi.org/10.1007/978-3-540-78800-3_24.

Müller, Peter (2002). *Modular Specification and Verification of Object-Oriented Programs*. Vol. 2262. Lecture Notes in Computer Science. Springer. ISBN: 3-540-43167-5. DOI: 10.1007/3-540-45651-1. URL: http://dx.doi.org/10.1007/3-540-45651-1.

Müller, Peter, Arnd Poetzsch-Heffter, and Gary T. Leavens (2003). "Modular Specification of Frame Properties in JML". In: *Concurrency and Computation: Practice and Experience* 15.2, pp. 117–154. DOI: 10.1002/cpe.713. URL: http://dx.doi.org/10.1002/cpe.713.

— (2006). "Modular Invariants for Layered Object Structures". In: *Sci. Comput. Program.* 62.3, pp. 253–286. DOI: 10.1016/j.scico.2006.03.001. URL: http://dx.doi.org/10.1016/j.scico.2006.03.001.

Naudziuniene, Daiva, Matko Botincan, Dino Distefano, Mike Dodds, Radu Grigore, and Matthew J. Parkinson (2011). "jStar-Eclipse: An IDE for Automated Verification of Java Programs". In: *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pp. 428–431. DOI: 10.1145/2025113.2025182. URL: http://doi.acm.org/10.1145/2025113.2025182.

Naur, Peter (1966). "Proof of Algorithms by General Snapshots". In: *BIT Numerical Mathematics* 6.4, pp. 310–316. ISSN: 1572-9125. DOI: 10.1007/BF01966091. URL: http://dx.doi.org/10.1007/BF01966091.

Nelson, Greg and Derek C. Oppen (1980). "Fast Decision Procedures Based on Congruence Closure". In: *J. ACM* 27.2, pp. 356–364. DOI: 10.1145/322186.322198. URL: http://doi.acm.org/10.1145/322186.322198.

Nielson, Flemming and Hanne Riis Nielson (1999). "Interprocedural Control Flow Analysis". In: *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, pp. 20–39. DOI: 10.1007/3-540-49099-X_3. URL: http://dx.doi.org/10.1007/3-540-49099-X_3.

Nielson, Flemming, Hanne Riis Nielson, and Chris Hankin (1999). *Principles of Program Analysis*. Springer. ISBN: 978-3-540-65410-0.

Nordio, Martin, Cristiano Calcagno, Bertrand Meyer, Peter Müller, and Julian Tschannen (2010). "Reasoning about Function Objects". In: *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, pp. 79–96. DOI: 10.1007/978-3-642-13953-6_5. URL: http://dx.doi.org/10.1007/978-3-642-13953-6_5.

Nordström, Bengt, Kent Petersson, and Jan M Smith (1990). *Programming in Martin-Löf's Type Theory*. Vol. 200. Oxford University Press Oxford.

O'Callahan, Robert and Daniel Jackson (1997). "Lackwit: A Program Understanding Tool Based on Type Inference". In: *Pulling Together, Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA, May 17-23, 1997*. Pp. 338–348. DOI: 10.1145/253228.253351. URL: http://doi.acm.org/10.1145/253228.253351.

O'Hearn, Peter W. (2005). "Scalable Specification and Reasoning: Challenges for Program Logic". In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pp. 116–133. DOI: 10.1007/978-3-540-69149-5_14. URL: http://dx.doi.org/10.1007/978-3-540-69149-5_14.

— (2012). "A Primer on Separation Logic (and Automatic Program Verification and Analysis)". In: *Software Safety and Security - Tools for Analysis and Verification*,

pp. 286–318. DOI: 10.3233/978-1-61499-028-4-286. URL: http://dx.doi.org/10.3233/978-1-61499-028-4-286.

O'Hearn, Peter W., John C. Reynolds, and Hongseok Yang (2001). "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, pp. 1–19. DOI: 10.1007/3-540-44802-0_1. URL: http://dx.doi.org/10.1007/3-540-44802-0_1.

O'Hearn, Peter W., Hongseok Yang, and John C. Reynolds (2004). "Separation and Information Hiding". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pp. 268–280. DOI: 10.1145/964001.964024. URL: http://doi.acm.org/10.1145/964001.964024.

Padhye, Rohan and Uday P. Khedker (2013). "Interprocedural Data Flow Analysis in Soot Using Value Contexts". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program analysis, SOAP 2013, Seattle, WA, USA, June 20, 2013*, pp. 31–36. DOI: 10.1145/2487568.2487569. URL: http://doi.acm.org/10.1145/2487568.2487569.

Park, Young Gil and Benjamin Goldberg (1992). "Escape Analysis on Lists". In: *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, pp. 116–127. DOI: 10.1145/143095.143125. URL: http://doi.acm.org/10.1145/143095.143125.

Parkinson, Matthew J. and Gavin M. Bierman (2005). "Separation Logic and Abstraction". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pp. 247–258. DOI: 10.1145/1040305.1040326. URL: http://doi.acm.org/10.1145/1040305.1040326.

Parkinson, Matthew J., Richard Bornat, and Cristiano Calcagno (2006). "Variables as Resource in Hoare Logics". In: *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pp. 137–146. DOI: 10.1109/LICS.2006.52. URL: http://dx.doi.org/10.1109/LICS.2006.52.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press. ISBN: 978-0-262-16209-8.

Plotkin, Gordon D. (2004). "A Structural Approach to Operational Semantics". In: *J. Log. Algebr. Program.* 60-61, pp. 17–139.

Polikarpova, Nadia, Carlo A. Furia, Yu Pei, Yi Wei, and Bertrand Meyer (2013). "What Good are Strong Specifications?" In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pp. 262–271. DOI: 10.1109/ICSE.2013.6606572. URL: http://dx.doi.org/10.1109/ICSE.2013.6606572.

Praun, Christoph von and Thomas R. Gross (2003). "Static Conflict Analysis for Multi-Threaded Object-Oriented Programs". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San*

Diego, California, USA, June 9-11, 2003, pp. 115–128. DOI: 10.1145/781131.781145. URL: http://doi.acm.org/10.1145/781131.781145.

Rakamaric, Zvonimir and Alan J. Hu (2008). "Automatic Inference of Frame Axioms Using Static Analysis". In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pp. 89–98. DOI: 10.1109/ASE.2008.19. URL: http://dx.doi.org/10.1109/ASE.2008.19.

Rémy, Didier and Jerome Vouillon (1997). "Objective ML: A Simple Object-Oriented Extension of ML". In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pp. 40–53. DOI: 10.1145/263699.263707. URL: http://doi.acm.org/10.1145/263699.263707.

Reps, Thomas W., Susan Horwitz, and Shmuel Sagiv (1995). "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pp. 49–61. DOI: 10.1145/199448.199462. URL: http://doi.acm.org/10.1145/199448.199462.

Reps, Thomas W. and Todd Turnidge (1996). "Program Specialization via Program Slicing". In: *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*, pp. 409–429. DOI: 10.1007/3-540-61580-6_20. URL: http://dx.doi.org/10.1007/3-540-61580-6_20.

Reynolds, John C. (1981). *The Craft of Programming*. Prentice Hall International series in computer science. Prentice Hall. ISBN: 978-0-13-188862-3.

— (2000). "Intuitionistic Reasoning about Shared Mutable Data Structure". In: *Millennial Perspectives in Computer Science*. Palgrave, pp. 303–321.

— (2002). "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pp. 55–74. DOI: 10.1109/LICS.2002.1029817. URL: http://dx.doi.org/10.1109/LICS.2002.1029817.

— (2005). "An Overview of Separation Logic". In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pp. 460–469. DOI: 10.1007/978-3-540-69149-5_49. URL: http://dx.doi.org/10.1007/978-3-540-69149-5_49.

Robert, Valentin and Xavier Leroy (2012). "A Formally-Verified Alias Analysis". In: *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, pp. 11–26. DOI: 10.1007/978-3-642-35308-6_5. URL: http://dx.doi.org/10.1007/978-3-642-35308-6_5.

Ruf, Erik (1995). "Context-Insensitive Alias Analysis Reconsidered". In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. PLDI '95. La Jolla, California, USA: ACM, pp. 13–22. ISBN: 0-89791-697-2. DOI: 10.1145/207110.207112. URL: http://doi.acm.org/10.1145/207110.207112.

Sabelfeld, Andrei and Andrew C. Myers (2003). "Language-Based Information-Flow Security". In: *IEEE Journal on Selected Areas in Communications* 21.1, pp. 5–19.

DOI: 10.1109/JSAC.2002.806121. URL: http://dx.doi.org/10.1109/JSAC.2002.806121.

Sagiv, Shmuel, Thomas W. Reps, and Reinhard Wilhelm (1999). "Parametric Shape Analysis via 3-Valued Logic". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999*, pp. 105–118. DOI: 10.1145/292540.292552. URL: http://doi.acm.org/10.1145/292540.292552.

Salcianu, Alexandru and Martin C. Rinard (2005). "Purity and Side Effect Analysis for Java Programs". In: *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Proceedings*, pp. 199–215. DOI: 10.1007/978-3-540-30579-8_14. URL: http://dx.doi.org/10.1007/978-3-540-30579-8_14.

Shapiro, Marc and Susan Horwitz (1997). "The Effects of the Precision of Pointer Analysis". In: *Static Analysis, 4th International Symposium, SAS '97, Paris, France, September 8-10, 1997, Proceedings*, pp. 16–34. DOI: 10.1007/BFb0032731. URL: http://dx.doi.org/10.1007/BFb0032731.

Sharir, M and A Pnueli (1978). *Two Approaches to Interprocedural Data Flow Analysis*. New York, NY: New York Univ. Comput. Sci. Dept." URL: https://cds.cern.ch/record/120118".

Shostak, Robert E. (1984). "Deciding Combinations of Theories". In: *J. ACM* 31.1, pp. 1–12. DOI: 10.1145/2422.322411. URL: http://doi.acm.org/10.1145/2422.322411.

Smans, Jan, Bart Jacobs, and Frank Piessens (2008). "VeriCool: An Automatic Verifier for a Concurrent Object-Oriented Language". In: *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings*, pp. 220–239. DOI: 10.1007/978-3-540-68863-1_14. URL: http://dx.doi.org/10.1007/978-3-540-68863-1_14.

— (2012). "Implicit Dynamic Frames". In: *ACM Trans. Program. Lang. Syst.* 34.1, 2:1–2:58. DOI: 10.1145/2160910.2160911. URL: http://doi.acm.org/10.1145/2160910.2160911.

Sozeau, Matthieu (2009). "A New Look at Generalized Rewriting in Type Theory". In: *J. Formalized Reasoning* 2.1, pp. 41–62. DOI: 10.6092/issn.1972-5787/1574. URL: http://dx.doi.org/10.6092/issn.1972-5787/1574.

Sozeau, Matthieu and the COQ development team (1997). *The Coq Proof Assistant Reference Manual: Version 8.6*. Inria.

Sridharan, Manu, Denis Gopan, Lexin Shan, and Rastislav Bodík (2005). "Demand-Driven Points-to Analysis for Java". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, pp. 59–76. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094817. URL: http://doi.acm.org/10.1145/1094811.1094817.

Strachey, Christopher (1967). *Fundamental Concepts in Programming Languages*. Lecture Notes, International Summer School in Computer Programming, Copenhagen. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.

Taghdiri, Mana, Robert Seater, and Daniel Jackson (2006). "Lightweight Extraction of Syntactic Specifications". In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006*, pp. 276–286. DOI: 10.1145/1181775.1181809. URL: http://doi.acm.org/10.1145/1181775.1181809.

Tip, Frank (1995). "A Survey of Program Slicing Techniques". In: *J. Prog. Lang.* 3.3. URL: http://compscinet.dcs.kcl.ac.uk/JP/jp030301.abs.html.

Vardi, Moshe Y. and Pierre Wolper (1994). "Reasoning about Infinite Computations". In: *Information and Computation* 115, pp. 1–37.

Volpano, Dennis M., Cynthia E. Irvine, and Geoffrey Smith (1996). "A Sound Type System for Secure Flow Analysis". In: *Journal of Computer Security* 4.2/3, pp. 167–188. DOI: 10.3233/JCS-1996-42-304. URL: http://dx.doi.org/10.3233/JCS-1996-42-304.

Wadler, Philip and R. J. M. Hughes (1987). "Projections for Strictness Analysis". In: *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, pp. 385–407. DOI: 10.1007/3-540-18317-5_21. URL: http://dx.doi.org/10.1007/3-540-18317-5_21.

Wand, Mitchell and William D. Clinger (1998). "Set Constraints for Destructive Array Update Optimization". In: *Proceedings of the 1998 International Conference on Computer Languages, ICCL 1998, Chicago, IL, USA, May 14-16, 1998*, pp. 184–195. DOI: 10.1109/ICCL.1998.674169. URL: http://dx.doi.org/10.1109/ICCL.1998.674169.

Wand, Mitchell and Igor Siveroni (1999). "Constraint Systems for Useless Variable Elimination". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pp. 291–302. DOI: 10.1145/292540.292567. URL: http://doi.acm.org/10.1145/292540.292567.

Weiser, Mark (1984). "Program Slicing". In: *IEEE Trans. Software Eng.* 10.4, pp. 352–357. DOI: 10.1109/TSE.1984.5010248. URL: http://dx.doi.org/10.1109/TSE.1984.5010248.

Wing, Jeannette M. (1987). "Writing Larch Interface Language Specifications". In: *ACM Trans. Program. Lang. Syst.* 9.1, pp. 1–24. DOI: 10.1145/9758.10500. URL: http://doi.acm.org/10.1145/9758.10500.

Xtext Documentation. https://eclipse.org/Xtext/. Accessed: 2016-09-11.

Zee, Karen, Viktor Kuncak, and Martin C. Rinard (2008). "Full Functional Verification of Linked Data Structures". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pp. 349–361. DOI: 10.1145/1375581.1375624. URL: http://doi.acm.org/10.1145/1375581.1375624.

Zhao, Yang and John Boyland (2008). "A Fundamental Permission Interpretation for Ownership Types". In: *Second IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE 2008, June 17-19, 2008, Nanjing, China*, pp. 65–72. DOI: 10.1109/TASE.2008.45. URL: http://dx.doi.org/10.1109/TASE.2008.45.

Zheng, Xin and Radu Rugina (2008). "Demand-Driven Alias Analysis for C". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: ACM, pp. 197–208. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328464. URL: http://doi.acm.org/10.1145/1328438.1328464.