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

École doctorale Matisse

présentée par

Martin Bodin

préparée à l'unité de recherche 6 074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
UFR Informatique Électronique (ISTIC)

Certified Semantics and Analysis of JavaScript

Thèse soutenue à Rennes
le 25 Novembre 2016

devant le jury composé de :

Sophie Pinchinat
Professeure / *Présidente*

Roberto Giacobazzi
Professeur / *Rapporteur*

Anders Møller
Professeur / *Rapporteur*

Philippa Gardner
Professeure / *Examinatrice*

Daniel Hirschkoff
Maître de conférence / *Examinateur*

Alan Schmitt
Directeur de recherche / *Co-directeur de thèse*

Thomas Jensen
Directeur de recherche / *Co-directeur de thèse*

Remerciements

J'ai eu la chance de pouvoir interagir avec un grand nombre de personnes durant cette thèse. Beaucoup m'ont énormément aidé tout au long de ce projet. Merci à tous.

J'aimerais tout d'abord remercier mes encadrants de thèse, Alan Schmitt et Thomas Jensen, sans qui cette thèse n'aurait jamais pu se faire. Ils ont su me donner de bons et utiles conseils. Il m'arrivait de ne pas les appliquer — j'ai d'ailleurs encore cette tendance à vouloir faire des preuves Coq avant d'avoir un prototype qui fonctionne, ou encore à faire des exposés un petit peu trop « sautillants ». Je leur remercie donc pour leur patience. Je leur suis aussi très reconnaissant de m'avoir poussé à présenter mes travaux dans divers ateliers et conférences. Les très nombreux cas particuliers dans la sémantique de JavaScript peuvent donner l'impression que l'on se bat contre une cause perdue ; Alan et Thomas ont pourtant réussi à me motiver pour pouvoir construire la thèse que voici !

I would like to heartfully thank Roberto Giacobazzi and Anders Møller for accepting to review my dissertation. My thesis covers various research fields—from abstract interpretation to separation logic, going through Coq proofs and the semantics of JavaScript—: it is a relief to have it reread by knowledgeable researchers from these different fields. I know that my writing is far from being perfect, and that some parts of this dissertation may appear too long or showing too many details. I am thus very grateful for their review.

I also thank all the other members of my jury: Sophie Pinchinat, Philippa Gardner, and Daniel Hirschkoff. It is a pleasure to know that my thesis has piqued their interest. I already had the pleasure to meet them on several occasions. I thank Daniel Hirschkoff for introducing me to functional programming at the École Normale Supérieure. I also thank Philippa Gardner for advising me during my internship in her working group: she helped me to find and express a lot of ideas from which the sixth chapter of this dissertation came out. I really appreciated how Sophie Pinchinat drove my defense: thank you!

I have had the chance to participate in various research projects during this thesis, in particular the projects JSCert, SecCloud, and AJACS. I would like to thank all of their members. Their meetings have helped exchange and seed various new ideas, which have proven to be very useful for this thesis. I would like to thank in particular Arthur Charguéraud, with whom I had a lot of fruitful interactions, all of which very interesting.

J'aimerais bien sûr remercier tous les membres de l'excellente équipe Celtique, qui ont dû me supporter pendant plusieurs années. C'est une excellente équipe, qui m'a permis de découvrir de nombreux points de vue très intéressants sur l'analyse de programme… mais aussi sur de nombreux autres thèmes tout aussi intéressants. Je remercie tout particulièrement les autres doctorants, André Oliveira Maroneze, Vincent Laporte, Pierre Wilke, Yannick Zakowski, Yon Fernández de Retana, Alix Trieu, Gurvan Cabon, mais aussi Pierre Karpman et les nombreux autres doctorants du centre pour leur discussions scientifiques ou non toujours très enrichissantes. Je remercie aussi chaleureusement Pierre Vittet pour m'avoir poussé vers l'usage régulier de l'espéranto dans ma vie de tous les jours. Un grand merci aussi à Laurent Guillo pour ses mots peu acratopèges, ainsi qu'à Charlotte Truchet pour ses problèmes de contraintes sur rendez-vous téléphoniques internationaux. Je remercie aussi François Schwarzentruber pour ses discussions toujours très originales, parfois à base de JavaScript et de chats quantiques.

I would like to thank all the members of the working group of Philippa Gardner in the Imperial College for their warm welcome. I really appreciated the various discussions with Anuj Sood. It has been a pleasure to be introduced to separation logic and its applications in the context of JavaScript by Gareth Smith, Daiva Naudžiūnienė, and Thomas Wood. I also thank Petar Maksimović and Teresa Carbajo Garcia for all their immensely useful help. This internship has been a great experience: thanks to all of these people.

J'aimerais aussi remercier le laboratoire IRISA, et ses conditions de travail excellentes. Un grand merci entre autres à Lydie Mabil, qui a réussi à protéger les divers doctorants du poids administratif si cher à l'administration française. Je remercie aussi toutes les personnes ayant participé aux « midi jeux » du mardi — entre autres Marwan Badawi, Carl-Johan Jørgensen, Julien Lefeuvre, Corentin Guichaoua, Matthieu Simonin, Alan Schmitt et tous les autres ! Ces pauses bienvenues ont contribué à la création d'une atmosphère chaleureuse dans le centre.

During this thesis, I had the chance to present my work into several workshops, such as JFLA, but also LTP or PLMW. I consider these experiences to be great opportunities for presenting my work to other researchers. I would thus thank all the organisers of similar workshops, which I think are very important for nowadays's research.

J'aimerais remercier tous mes amis et ma famille proche, qui m'ont beaucoup encouragés durant ma thèse. Un grand merci donc à Arnaud de Latour, Hugo Martin, ainsi que mes sœurs et mes parents, pour m'avoir poussé à continuer dans mes recherches malgré leur complexité. Je remercie en particulier Clara Bodin pour m'avoir aidé à trouver la citation du cinquième chapitre de cette thèse. Une pensée pour feu ma cousine Élise Troël, dont je conserve un agréable souvenir d'une discussion à propos de cette thèse.

Mia lasta sed ĉefa dankego venas al la amo de mia vivo, Kadígia Constantino Pinheiro de Andrade. Ŝi multege helpis min elteni la malĝojajn momentojn de mia vivo, kaj anstataŭigi tiujn per bonaj kaj trankvilaj sentoj. Mi dankegas ŝian apogon dum mia tuta doktoriĝo, ŝian subtenon kiam mi devis malfrue kaj daŭre labori, kaj ŝian ĉiaman bonhumoron kaj buntemecon. Senliman dankegon al ŝi!

Contents

List of Figures

List of Programs

Résumé en Français

Il est devenu courant de faire confiance à des logiciels dans nos sociétés. De nombreuses personnes possèdent au moins un appareil — souvent un téléphone portable — contenant des informations privées, comme leurs contacts ou leurs courriels. Ces appareils sont souvent équipées de microphones et de divers moyens de connaître leur position. De manière similaire, de nombreuses personnes donnent des informations privées à des sites web ; la popularité actuelle des réseaux sociaux en est l'exemple le plus marquant. Nous faisons confiance aux logiciels sous-jacents. En particulier, nous considérons que ces logiciels ne nous espionnent pas, ou qu'ils ne donnent pas d'information privée à n'importe qui. Pourtant, la question se pose : les programmes présents dans les téléphones ou les pages web modernes sont très complexes. Ils sont souvent composés de plusieurs composants provenant de différentes sources, toutes n'étant pas de confiance.

Ces programmes sont souvent écrits dans un language de programmation très dynamique : JavaScript. Le but initial de JavaScript était de rendre les pages web interactives. Il a été conçu pour aider le prototypage logiciel (la création rapide de prototypes logiciels). La sécurité n'était alors pas une préoccupation importante des concepteurs du langage de programmation. Entre-temps, JavaScript a gagné en popularité tant et si bien qu'on l'utilise maintenant pour concevoir des logiciels manipulant des données sensibles.

Il est important de pouvoir répondre à cet usage en proposant diverses manières de pouvoir évaluer la confiance de logiciels. La méthode la plus utilisée est le test : le logiciel, ainsi que tous ses composants, sont exécutés sur de nombreux cas particuliers ; leur comportement est alors comparé avec le comportement attendu. Tester un programme permet de repérer des bogues (des comportements inattendus), ou des failles de sécurité, mais ne permet pas de prouver leur absence. Certains bogues ont été découverts bien après que le logiciel en faute soit largement utilisée. Un exemple impressionnant est celui d'Heartbleed [Dur+14] : bien que le logiciel OpenSSL soit très utilisé et que son code source soit disponible à tous, cette faille de sécurité a été découverte plus de deux ans après son introduction. Ceci est d'autant plus impressionnant qu'OpenSSL est utilisé dans des contextes où la sécurité est très importante et est donc très testé. Le test a donc de grandes limitations dans sa capacité à repérer des bogues et des failles de sécurité.

Les méthodes formelles visent à fournir une preuve mathématique du bon fonctionnement d'un programme. Ceci a l'avantage de considérer tous les cas : contrairement au test, il n'est pas possible de se retrouver lors d'une exécution dans une situation non prévue par la preuve. En contrepartie, la construction d'une preuve de programme est très souvent complexe, longue et fastidieuse — en particulier pour des langages tels que JavaScript. Pour pouvoir faire confiance à de telles preuves, nous utilisons des assistants de preuves tels que Coq [CHP+84]. Ces outils ont été conçu avec soin, de telle sorte que l'on puisse faire confiance aux preuves qu'ils acceptent. Ils peuvent être aussi très « sceptiques », rendant la construction de preuve assez complexe (voir un exemple en partie 5.3). Ces outils sont la plupart du temps basés sur un langage de tactiques permettant d'automatiser une partie de la construction des preuves : ces tactiques construisent des termes de preuves vérifiés après construction par l'assistant de preuve.

Pour pouvoir ne serait-ce qu'énoncer dans un assistant de preuve qu'un programme s'exécute toujours correctement quels que soient ses entrées et son environnement d'exécution nous avons besoin de définir la sémantique du programme, ou plus généralement la sémantique de son langage de programmation. Il existe plusieurs façons de spécifier la sémantique d'un langage de programmation. Nous utilisons ici une approche dite en grand pas : nous définissons un prédicat $\Downarrow$ décrivant l'exécution d'un programme. Le triplet sémantique $\sigma, p \Downarrow r$ exprime que le programme p s'exécutant dans le context σ peut donner le résultat r. Ce prédicat est défini par induction à partir de règles d'inférences. Le théorème de correction du programme p est de la forme suivante : pour tout triplet sémantique $\sigma, p \Downarrow r$ le résultat r est un résultat attendu du programme p. Une étape importante de cette thèse a consisté à exprimer la sémantique de JavaScript sous cette forme.

Le projet JSCert vise à précisément spécifier JavaScript dans l'assistant de preuve Coq. Le projet a impliqué 8 personnes pendant un an. La sémantique complète de JSCert contient plus de 900 règles d'inférences. Devant une telle complexité, de nombreux moyens ont été mis en œuvre afin de pouvoir faire confiance en la sémantique de JSCert. D'une part, JSCert s'appuie sur la spécification officielle de JavaScript: ECMAScript. Les structures de données manipulées par les deux sémantiques sont identiques, et JSCert utilise les mêmes étapes de calcul qu'ECMAScript. Toute ligne de la spécification correspond à une règle d'inférence de JSCert. Ceci donne un avantage intéressant à JSCert: il est possible de facilement le mettre à jour pour s'adapter à des changements d'ECMAScript en modifiant de manière similaire JSCert. Plus de détail est donné sur le rapprochement entre JSCert et ECMAScript en partie 2.7.2. D'autre part, JSCert est muni d'un interpréteur, JSRef. Ce dernier est certifié correct vis-à-vis de JSCert: si l'interpréteur renvoie un résultat, ce dernier est en accord avec la sémantique de JSCert. Nous avons pu exécuter JSRef sur des suites de tests de JavaScript, et vérifier la conformité des résultats. Cette double vérification de JSCert avec JavaScript, tant au niveau de sa spécification officielle ECMAScript, que des suites de tests, en fait la sémantique formelle la plus fiable de JavaScript à l'heure actuelle. Cette sémantique permet la certification de divers outils formels basés sur JavaScript, en particulier la preuve de programmes JavaScript.

La preuve de programme est longue et fastidieuse, surtout pour des sémantiques de cette taille. Plutôt que de prouver des programmes à la main, nous avons choisi de construire un analyseur que nous prouverons correct. Cette approche permet l'analyse systématique de programme, quelle que soit leur taille. Pour cela, nous nous appuyons sur le formalisme de l'interprétation abstraite [CC77a]. Ce formalisme propose d'exécuter des programmes en remplaçant les valeurs concrètes précises par des valeurs abstraites moins précises. Par exemple, ou peut abstraire les valeurs concrètes 18 et 42 par la valeur abstraite $+$, et la valeur -1 par $-$: cette abstraction approxime chaque valeur concrète par son signe. Le choix du domaine abstrait détermine les propriétés que l'on souhaite détecter dans le comportement d'un programme. Abstraire les valeurs par leur signe est rarement suffisant ; les octogones [Min06b] fournissent un exemple plus complexe de domaine abstrait.

L'interprétation abstraite se divise en deux étapes. D'abord, une sémantique abstraite est définie et prouvée correcte. Cette sémantique abstraite est la plupart du temps non déterministe — elle autorise par exemple de perdre en précision le long de la dérivation. Ensuite, un analyseur est défini et prouvé correct vis à vis de cette sémantique abstraite. Cet analyseur utilise le non-déterminisme de la sémantique abstraite pour mettre en œuvre des heuristiques. Il pourra par exemple choisir de simplifier une valeur abstraite $v_1^\sharp$ au cours du calcul par une valeur abstraite $v_2^\sharp$ moins précise. Ce choix peut être pertinent par exemple lorsque la valeur abstraite $v_1^\sharp$ prend trop de place en mémoire. Ces deux étapes (construction de la sémantique abstraite, puis d'un analyseur) sont souvent confondues pour des raisons pratiques. Par exemple, il arrive que la sémantique abstraite utilise des heuristiques — typiquement des techniques d'accélération de convergence par opérateur d'élargissement (dits de *widening*). Une telle sémantique abstraite est déterministe, et n'est donc compatible qu'avec l'analyseur pour lequel elle a été définie. Dans le cas de JavaScript, la sémantique considérée est immense : plus de 900 règles de réduction, sans compter la bibliothèque par défaut. À titre de comparaison, le langage de programmation analysé par l'analyseur certifié Verasco [Jou+15], le C♯minor, contient moins de 50 règles de réduction. Dans le cas de JavaScript, il n'est donc pas réaliste de fusionner les deux étapes que sont la construction et la preuve d'une sémantique abstraite et celles d'un analyseur.

Une contribution importante de cette thèse a consisté à formaliser en Coq le processus de construction d'une sémantique abstraite — qui était jusqu'alors appliqué au cas par cas. Nous avons proposé un procédé permettant d'abstraire de manière systématique chaque règle de réduction, sans avoir à comprendre comment les règles concrètes interagissent entre elles. Nous nous basons sur les travaux de Schmidt (voir partie 3.3). Ce procédé consiste à abstraire *indépendamment* chaque règle concrète pour former une règle abstraite — ce qui requérait de comprendre comment la sémantique fonctionne. En particulier, les règles concrètes ne sont pas fusionnées pour prendre en compte leurs interactions. Ce dernier point contribue grandement à systématiser l'abstraction d'une sémantique. Chaque règle abstraite ainsi construite partage avec sa règle concrète les même caractéristiques syntaxique : sa structure, les termes auxquels elle fait référence, ainsi que son nom de règle. Seuls sont modifiés les fonctions de transfert et ses conditions d'application.

Nous avons précisément défini chacun des éléments constitutifs d'une règle de dérivation (sa structure, ses fonctions de transfert, etc.). Nous nous sommes fortement appuyés sur les restrictions structurelles de JSCERT dites en *sémantique à bond* (ou en *pretty-big-step*).

À partir de ces règles abstraites, nous pouvons construire de manière générique une sémantique abstraite. La manière de construire cette sémantique abstraite diffère en trois points de la sémantique concrète. D'abord, au lieu de n'appliquer qu'une seule règle à chaque étape de calcul, nous devons considérer toutes les règles qui s'appliquent : les dérivations se divisent pour explorer chaque cas indépendamment. Ensuite, les dérivations abstraites peuvent être infinies, ce qui donne une sémantique aux boucles par point fixe (voir partie 4.4.2.2) — cette technique est fréquemment utilisée pour l'analyse de boucle en interprétation abstraite. Enfin, des règles intermédiaires, dites *non-structurelles*, sont appliquées à chaque étape. Ces règles permettent d'appliquer divers types de raisonnement qui ne rentrent pas tel quel dans notre formalisme. Nous avons prouvé en COQ qu'étant donné un certain nombre de contraintes *locales* sur les règles de réduction abstraites, ainsi que sur les règles non structurelles, la sémantique abstraite (globale) est correcte : pour tout triplets sémantiques concret $\sigma, p \Downarrow r$ et abstrait $\sigma^\sharp, p \Downarrow^\sharp r^\sharp$ tels que les entrées concrète σ et abstraite $\sigma^\sharp$ correspondent, les résultats r et $r^\sharp$ correspondent aussi.

Nous avons construit un domaine abstrait pour un langage similaire à JAVASCRIPT dans ce formalisme. Ce domaine est basé sur la logique de séparation [IO01 ; ORY01 ; Rey02 ; Rey08]. Cette logique est connue pour ne pas bien interagir avec l'interprétation abstraite : nous l'avons choisi pour évaluer la généralité de notre formalisme. La règle de contexte — une règle centrale en logique de séparation — s'inscrit naturellement dans le cadre des règles non-structurelles. Cette règle permet de focaliser une dérivation sur les ressources manipulées par le programme analysé ; typiquement en ignorant des ressources inutiles. Cette règle permet entre autres de rendre l'analyse de programme modulaire. Nous avons pu identifier très précisément pourquoi la logique de séparation et l'interprétation abstraite interagissent de manière inattendue : l'interprétation abstraite permet de renommer les identifiants utilisés dans les formules logiques tant que cela ne change pas leur concrétisation. Ceci entre en conflit avec la règle de contexte, qui nécessite que les identifiants soient cohérents tout au long d'une dérivation. Nous avons introduit la notion de *membrane* pour propager ces renomages le long des dérivations. La formalisation en COQ de ce domaine est encore en cours (voir partie 6.4.5), mais il offre déjà une solution prometteuse pour construire dans notre formalisme un domain mêlant interprétation abstraite et logique de séparation.

Les contributions de cette thèse sont donc triples. Le projet JSCERT a permi de construire une sémantique formelle de confiance pour le langage JAVASCRIPT. Nous avons fourni un formalisme générique pour construire des sémantiques abstraites à partir de sémantiques concrètes telle que JSCERT. Enfin, nous avons construit un domaine non trivial pour ce formalisme. Il est maintenant possible d'instancier ce domaine à JSCERT, ce qui produira une sémantique abstraite certifiée de JAVASCRIPT, permettant la certification d'analyseurs.

Introduction

Le but de cette thèse est de munir son auteur du titre de Docteur.

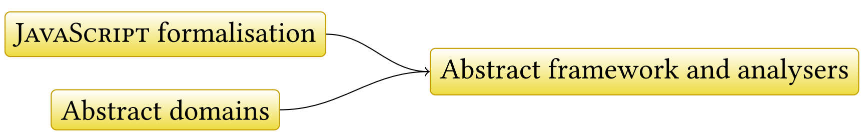
Michèle Audin [Aud97]

Software is everywhere, from the servers influencing our economy to the microcontrolers of our vehicles. Some devices, such as smartphones, follow most of our moves and have access to private information such as emails or meetings. Software can be unnoticed, but it raises an important security issue. Security flaws are regularly discovered in software, Heartbleed [Dur+14] being one of the most impressive example. In this dissertation we are interested in particular to the JavaScript language. This language is used in a variety of places, from webpages to smartphones applications, and is a good target for analyses.

Guaranteeing some security properties is a difficult task. Formal methods offer an elegant solution to this problem: they have proven their abilities to find various types of bugs without having to run any JavaScript programs [Pol+11a; MT09a; MMT10]. Unfortunately, JavaScript is a particularly complex programming language, and building formal tools for JavaScript requires a significant effort. To trust the results of an analyser, it needs to have passed some form of certification. But JavaScript certification is hindered by the numerous corner cases of JavaScript's semantics. Building analysers for JavaScript is already a difficult task, building *trustable* ones is even more difficult.

This thesis aims at reducing the gap between formal methods and JavaScript. In particular, it aims at building techniques to create and prove analysers for the JavaScript programming language. This thesis has led—with the help of the other members of the JSCert project—to the construction of the JSCert formal specification of JavaScript. Although recent, the applications of this formalisation are already very promising.

This dissertation is divided into three parts: the JSCert formalisation of JavaScript, the presentation of techniques to build certified JavaScript analysers from this formalisation, and the abstraction of JavaScript memory model.



Chapter 1 presents the JavaScript language; including Sections 1.2.4 and 1.2.5 which present the parts hindering analyses. This chapter also presents JavaScript memory model in Section 1.2.3, which will be useful to understand Chapters 2 and 6. The first contribution appears in Chapter 2, which focusses on the JSCert specification of JavaScript. This specification has been defined to be trusted, and thus to serve as a basis for the construction of trusted analysers, such as the ones built as part of this thesis. The JSCert specification has been built as a joint work with the other members of the JSCert project.

The large size of JSCert raises the question of how practical are formal methods on such a large semantics. Chapter 3 introduces the framework of abstract interpretation [CC77a] in the context of the Coq proof assistant [CHP+84]. Chapter 3 does not present any new contribution, but presents important notions to understand the following chapters. Chapters 4 and 5 present how to build certified analysers for large semantics, such as JSCert. The presented method is a novel way to build analysers sound by construction: their proof of soundness have been designed to be scalable and adaptable to the changes of JSCert. Chapter 4 exploits this basic idea and presents its Coq formalisation. Chapter 5 extends this formalisation to structural rules, a key point to built practical analysers.

The analysers of Chapters 4 and 5 are parametrised by an abstraction of the memory model (and by abstract domains in general). Chapter 6 presents an instantiation of this framework to the memory model of JavaScript. This chapter uses separation logic [IO01; ORY01; Rey02; Rey08] as a basic starting point, as it has already proved to be suitable for JavaScript's memory model [GMS12]. Separation logic and abstract interpretation have slightly different hypotheses on the way domains should be defined, and it is interesting to see how they interact in this framework. ⌜Chapters 4, 5, and 6⌝ focus on small languages and not on JSCert: they only aim at showing the developed techniques, and how scalable they are. The application of these techniques to JSCert is left as a further work.

This thesis is accompanied by a webpage [Bod16] containing links to the programs presented in this dissertation, in particular, runnable versions of the different analysers built during this thesis. I strongly encourage the reader to try running the presented JavaScript programs in various environments, as well as testing the presented analysers.

Finally, a note on the presence of the symbols '⌜' and '⌝' in this dissertation. These symbols are directly inspired from a proposition by Madore [Mad15] to disambiguate English text. They should be considered as grouping symbols, for instance helping to make the difference between "dynamic ⌜program analysis⌝" and "⌜dynamic program⌝ analysis"—a common ambiguity in ⌜research paper⌝ titles. These symbols considerably help my parsing of English sentences, but I understand that some people find them distracting. I have thus tried to make these symbols as discrete as possible.

1 The JavaScript Language

La ord' en la vortaro plaĉas al mi.
La sistematiko plenigas min per ĝuo.
Tamen el la vortoj kiujn mi trovis en ĝi,
la plej bela estas "tohuvabohuo".

Nanne Kalma [Kal13]

This chapter aims at presenting the JavaScript programming language. Section 1.1.1 provides a quick introduction to how JavaScript has evolved, then Sections 1.1.2 and 1.1.3 present some uses of JavaScript, some of which being critical applications. Section 1.2 aims at giving some insights about JavaScript's semantics, in particular its memory model in Section 1.2.3. The goal of Sections 1.2.4, 1.2.5, and 1.2.7 is to show some difficulties associated with the language's semantics, in particular when building analysers.

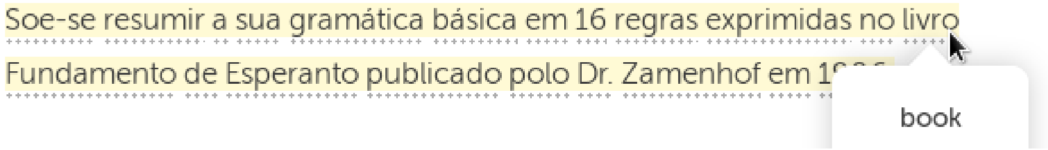
1.1 Presentation of JavaScript

Knowing the context in which JavaScript was created helps understanding some of its peculiarities—in particular, its usages shifted along the years. We start with a short introduction on how JavaScript has been created as well as some of its current applications.

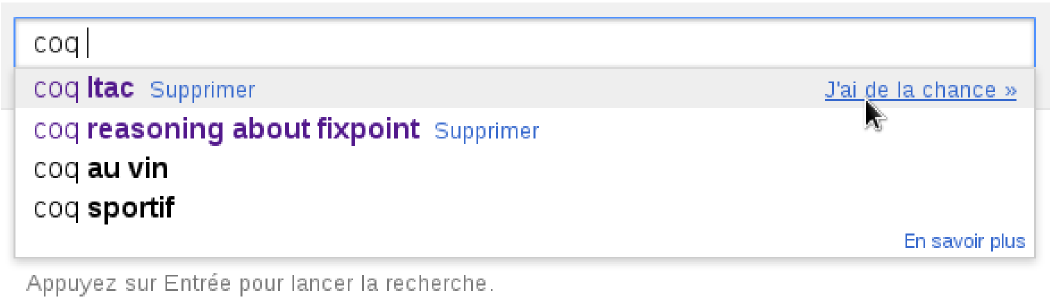
1.1.1 A Quick History of the Language

JavaScript began in 1995, when Brendan Eich was asked to build a scripting language for Netscape. The idea was to build a language looking like Java whilst being light enough to appeal to non-professional programmers. The target usage of this language was to enhance webpages by adding interactivity through client-side scripts: Figure 1.1 shows examples of various websites using JavaScript to provide client-side interactivity. To interest the broadest number of potential programmers, features from various programming languages were added: the language mixes features from functional and object oriented programming languages; as well as some features of Perl. Due to very short release dates, the initial version was written in ten days.

In the following years, JavaScript was adopted by other web browsers, each adding or adapting some of its features. ECMAScript then took care of the standardisation effort, and released the first specification of JavaScript in 1997. The third version of the specific-



(a) Duolingo (`http://www.duolingo.com`)



(b) Google (`http://www.google.com`)

(c) The companion website of this thesis [Bod16]

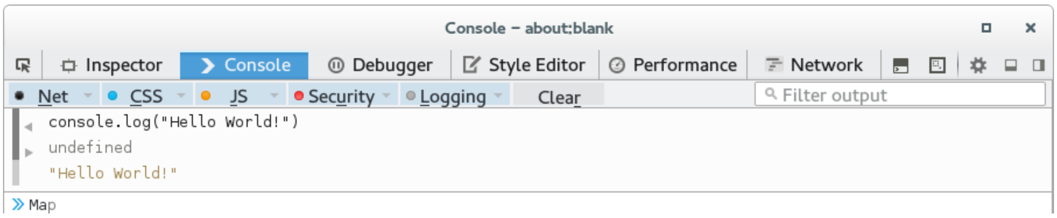
Figure 1.1: Examples of webpages enhanced by JavaScript

ation, ECMAScript 3, was considered to be the main version of JavaScript for approximatively ten years, before the recent new 5th and 6th versions came to life. JavaScript now continues to receive updates as new versions of ECMAScript are coming out.

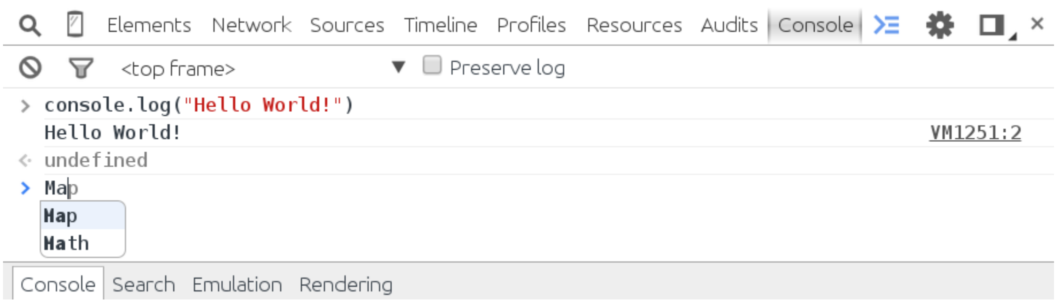
The standard(s) of JavaScript is a defining feature of the language. Technically, the language defined by the ECMAScript standard is "ECMAScript", but as in practise the language is known by the whole community as JavaScript, I shall consider that ECMAScript refers to the language specification. The main goal of the ECMAScript specification is to provide interoperability, in particular to avoid each actor of the JavaScript community to have slightly different notions of how JavaScript behaves: the ECMAScript standard does not focus on defining a principled language (see Section 1.2.1), but on creating a consensus on what JavaScript is.

1.1.2 Where is JavaScript Used?

Although JavaScript started as a small scripting language only aimed to be used in browsers for non-critical software, it is now used in a variety of places, some of which far from the original target of the language. Figure 1.2 shows some examples of JavaScript consoles—respectively in Mozilla Firefox, Chromium, and Node.js. Mozilla Firefox uses the JavaScript engine SpiderMonkey, and the other two consoles use V8. We can already see with the example of Node.js that JavaScript escaped from its original envir-



(a) In Mozilla Firefox



(b) In Chromium

(c) In Node.js

Figure 1.2: Different JavaScript consoles

onment: Node.js executes JavaScript programs in a Unix environment; it is used in some websites to run JavaScript in the server-side, for instance to help code factorisation in the server and client sides.

JavaScript has imposed itself as being *the* (as opposed to *one*) programming language of the Internet. A lot of devices (such as phones or e-book readers) nowadays can access the Internet, and are able to execute JavaScript. One example of this is the recent introduction of Chromebooks, whose (almost) only feature is to browse the Internet. JavaScript being executable in almost every device, it has become a programming language simple to manage: as it can be executed everywhere, few costs are needed to adapt it to another architecture, device, or environment. This has led to consider JavaScript as a target language for development, but also for compilation. There now exist compilers from various languages to JavaScript, such as Js_of_ocaml [VB14] compiling OCaml programs, and Emscripten [Zak11], compiling LLVM programs into JavaScript.

Compiling a program into JavaScript is interesting for manageability, but the performance of an interpreted program is usually bad compared to a compiled one. This statement tends to fade with the current advances of just-in-time compilation. Recent JavaScript interpreters no longer interpret, but compile programs with static or dynamic optimisations. Initiatives such as ASM.JS [HWZ13] also helped increasing the speed of JavaScript programs. Nowadays, most JavaScript programs run without slowness.

Compilers to JavaScript enable developers to write programs in other programming languages (usually statically typed) whilst executing them in a browser or other JavaScript environments. This approach has also been used in industry to provide more guarantees to JavaScript programs; for instance Typescript [BAT14] is a variant of JavaScript with partial typing. Some of these compilers may assume that the program is the only program in the environment; this is usually not the case, in particular in mash-ups.

1.1.3 Mash-ups

Mash-ups refer to webpages built on top of several external resources, which interacts with each other. For instance, let us imagine a webpage looking for hotels in a given area; such a webpage could include a lot of external resources:

- a search engine, to search for hotels in the area,
- a map, to display where all these hotels are located,
- a calendar, to display when rooms are available,
- some social network plug-ins,
- a machine translation plug-in, for users who do not know the local language(s),
- some advertisements.

All these third party programs are JavaScript programs loaded from different sources. Some of these manipulate sensitive information (such as the user's travel dates), whilst some are not trusted (at least the advertisements should be in this category).

To have an idea about what kind of programs can be in a mash-up, I recommend to check the list [Goo02] of API provided by Google: it gives an idea of the variety of programs which developers can import from a third party. JavaScript mash-ups are easy to create: one just has to include these third party scripts. Such included scripts are also usually written to have very general applications, including ones which developers did not necessarily think about. This imaginary hotel searching webpage may be usable without trouble in a smartphone, even if the developers did not thought about this. For a personal example, the JavaScript variable `navigator.language` of my browser is set to `"eo"`; as a consequence various websites—including some probably written by English-speaker-only people—display a login page (partially) in Esperanto.

The problem with this practise is security: when JavaScript programs are imported from various sources in the same webpage, these programs are executed in the same environment. As a consequence, the isolation of sensitive data is not an easy task. There exist libraries, such as ADSafe [Cro08], restricting programs to some boundaries; these libraries sometimes rely on JavaScript analysers. However these libraries have not yet been formally proven to be safe, and bugs or other weaknesses can be found [Pol+11a; MT09a; MMT10]. We are thus in the need of certifying some JavaScript analysers. This thesis aims at applying standard formal method techniques to produce certified analysers.

1.2 Presentation of the Language Semantics

In contrary to many other scripting languages, JavaScript is defined by a precise specification, the ECMAScript standard [ECM99; ECM11]. This standard exists in several versions, and not every browser is up-to-date concerning new features. The current version is the 6th, which added many features from the 5th version. In practise, most web browsers implement every features of ECMAScript 5 as well as some of ECMAScript 6. Some also provide non-standardised features, such as some special behaviours of the `__proto__` field (see Section 1.2.3). I will mainly focus about ECMAScript 5 in this document. More about the style of the ECMAScript specification can be found in Section 2.4.

I do not intend to give a full specification of JavaScript semantics in this document, but I will try to explain everything needed to understand the issues with JavaScript analyses. The two main difficulties of JavaScript come from the scoping of variables and fields, and from the size of its specification. I am thus going to focus mainly on the object model (which defines how variables and fields are scoped) and type conversions (which form a large part of the specification). However, before starting to present the semantics, I will recall a basic human principle—which I often forgot—when dealing with JavaScript.

1.2.1 Please, Do Not Criticise JavaScript Eagerly

At a look of JavaScript's semantics, it is easy to say that the language is not suitable for most applications. Many people from our research community (including me)—along with other communities [LLT10]—have been eager to state that JavaScript is a *bad* language. After working with JavaScript for more than three years, I sure can state that JavaScript is *unprincipled*. I would however like to emphasize that saying that JavaScript is bad may be offensive to the ECMAScript community.

JavaScript has many unexpected exceptions and pitfalls. Mosts come from retrocompatibilities of the language: the Internet is not a unified place and breaking the web by changing the behaviour of a widely used construct is not an option. Most of the people working with JavaScript do not really have the choice of their language. As researchers, we often create prototypes from scratch, allowing us to use any language, such as OCaml or Haskell (which can also have some surprising behaviours [JL14]). In industry, updating huge amount of code costs a lot of money, and inertia can be a valid choice.

I like to compare JavaScript with the usage of English in scientific works. English can be a beautiful language to write poetry, but its difficulty to read, pronounce, learn, as well as its frequent ambiguities [Mad15] makes it a pessimal choice for scientific communications.

There are some proposed solutions, like Esperanto [PGG; Cor12; Max88] (to reduce the difficulty to read and learn) or Lojban [Cow97] (to reduce language ambiguities). In both the JavaScript and English cases we are fighting against inertia. As this illustration is not always taken seriously and because I am biased on this matter, let us use another metaphor: cigarette lighter multipliers, which provide several plugs from a single cigarette lighter plug. They are surprisingly easy to find in highway shops. Of course, their existence is not due their literal usage (lighting several cigarettes in a row), but merely that cigarette lighter plugs became a standard and escaped from its original usage.

JavaScript became a widespread language for a variety of usages for which it has never been designed. Once it became a language runnable by any computer and device, industry quickly adapted and switched to this language. In this way, industry is no longer forced to deal with several versions of their programs, one for each operating system and environment. The success of JavaScript solved the problem of interoperability, but added the problem of the JavaScript language itself. There is however an increasing awareness on this matter, and many people are now looking for solutions to improve the language safety. For instance, the strict mode of JavaScript solved the problem of lexical scoping (see Section 1.2.8), and new features of JavaScript are examined through a scrutiny process before acceptance [ECM15]. Features which are fixable and not used a lot are fixed; for instance the semantic of loops in ECMAScript 5 was incompatible with loop unrolling because of some corner cases, but (almost) no JavaScript program relied these on corner cases: this feature has thus been fixed as soon as it was noticed [Theb]. To conclude, I would thus like to present JavaScript as it is now, as a fact, and not as a way to criticise people and historical accidents which have made JavaScript as it currently is.

1.2.2 Basics

The grammar of JavaScript is divided into three main categories: expressions, statements, and programs. A JavaScript program consists of a list of statements; programs can be found at top level, but also as a function body. Similarly, the argument of the `eval` function, once parsed, is a JavaScript program. Apart from some corner cases (such as those presented in Section 1.2.7), JavaScript's syntax should be straightforward for people familiar with other C-like programming languages.

There are six distinct types of values in JavaScript, in ECMAScript 5:

- locations, which are described in detail in the next section. They can be seen as pointers pointing to JavaScript's objects.
- Numbers, which follow the IEEE 754 double-precision float specification [Ste81; Ste+85]. This includes the particular numbers `NaN`, `+Infinity`, and `-Infinity`.
- Unicode strings, encoded in UTF-16 [Con16].
- the two booleans `true` and `false`.

- The value `null`. Note that it is not a location, but a stand-alone value; in particular, comparing it to the value `0` (through the expression `null === 0`) returns `false`.
- the value `undefined`. This is a defined value: notice the typographic and grammatical differences between the value `undefined` and an undefined value.

There are no integers: (floating-point) numbers are used instead. Values other than locations make sense independently of the heap: they are called *primitive values*. Variables are not typed: the keyword to introduce a variable is `var` (without type annotation), and a given variable can receive any kind of value. This yields complex type conversions described in Section 1.2.4. But let us first focus on the memory model.

1.2.3 Object Model

This section describes how JavaScript's memory model works. Section 1.2.3.1 describes how variables and object's fields are looked up and Section 1.2.3.2 describes how they are manipulated during execution. The manipulations shown here do not always preserve scopes, which can be counter-intuitive for a programming language. There is however a built-in sublanguage of JavaScript designed to preserve them (see Section 1.2.8).

1.2.3.1 Field and Variable Look-up

JavaScript programs manipulate a *heap*. Objects in the heap are indexed by *locations*. Intuitively, locations can be seen as pointers. In contrary to C, there is no pointer arithmetic in JavaScript: the only things we can do with locations is to compare them (through equality) or to access their object. There is no way to dispose of (or deallocate) objects in JavaScript. By this way, the language is guaranteed to never fault because of dangling pointers. Naturally, real-world JavaScript interpreters perform garbage collection, but it should never interfere with the semantics.

Objects are maps from fields (named *properties* in the ECMAScript standard) to values. Fields can be added or deleted at will[1] during execution. Every object has an implicit prototype in the form of a special field which we call $@proto$; its value is either `null` or a location. Although this is not standardised in ECMAScript 5 (but is in ECMAScript 6 through the `Object.setPrototypeOf` function), most JavaScript interpreters enable programs to access implicit prototypes through the `__proto__` field. Unless a prototype is `null`, its pointed object also has an implicit prototype, and so on, forming a prototype chain. The semantics of JavaScript guarantees that no loop can form in a prototype chain.

Intuitively, the field $@proto$ of a location l points to a location representing the class from which l inherits: each time a field `f` of a location l is looked up, l is checked to effectively have this field; if not, the prototype chain is followed until such an `f` is found. The

[1] There are ways to prevent some fields to be changed—for instance `Object.seal`, prevents further deletion or addition of fields for the given object—but we shall not focus on these techniques.

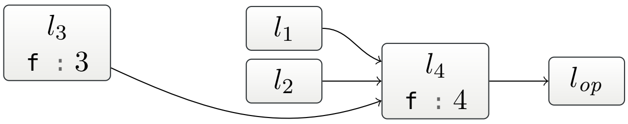


Figure 1.3: Illustration of a prototype chain

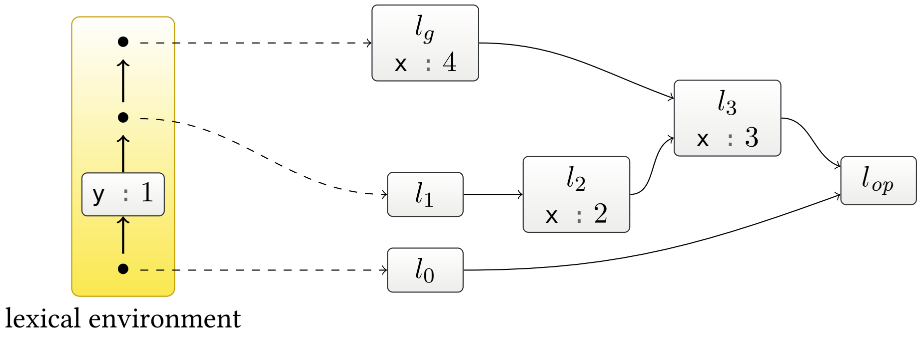
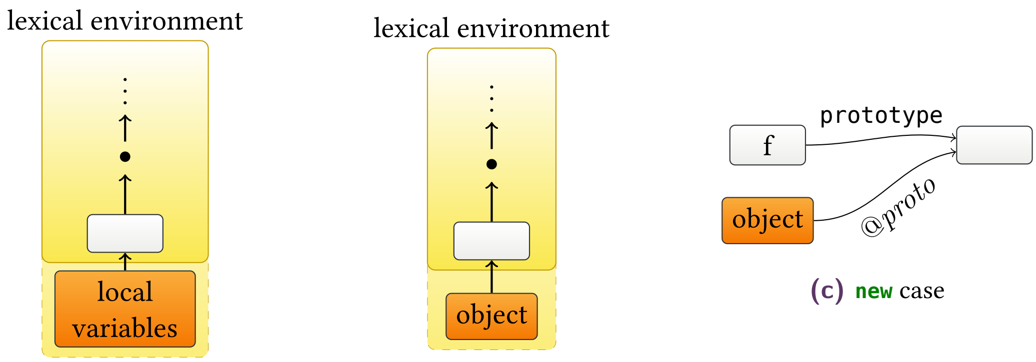


Figure 1.4: A JavaScript lexical environment

fields present in a prototype chain are thus common to all the objects with this prototype chain, as would the methods of a class in an object-oriented programming language. When evaluating an expression of the form `e.f`, the expression `e` is evaluated and should result as a location l; then `f` is looked up. In case no value is found, the value **`undefined`** is returned. Figure 1.3 represents such a prototype chain, plain arrows depicting implicit prototypes; the access to the field `f` returns 3 at locations l_1, l_2, and l_4, but 4 at location l_3, and **`undefined`** at location l_{op}. Some fields are also defined by getters and setters—which are functions called when accessing them—but we shall not detail them.

The execution context of a JavaScript program comprises a *lexical environment* (called *scope chain* in ECMAScript 3). It intuitively corresponds to a call stack. The lexical environment is a stack of *environment records*. Environment records can be either *declarative* or *object* environment records. The former is typically created when calling a function; it is a mapping from identifiers to values. The latter may be surprising: it is a location (called *scope* when in this position). All the fields of the associated object are then considered as being directly in the context as variables.

The top of the lexical environment stack is an object environment record with a special location l_g referring to the *global object*: global variables reside in this object. When looking up the value of a variable `x`, it is searched in the lexical environment. More precisely, the value of `x` will be found in the first environment record in the lexical environment where it is defined. This behaviour is similar to the one of a lexical scope (local variables having priority over global variables of the same name). However, as object environment records are usual objects, they can be dynamically modified. Moreover, we see below that scopes can be manually added to the chain using the **`with`**-construct. Variable look-up is also determined by the *prototypes* of the objects under consideration.



(a) Function call case **(b)** `with` case **(c)** `new` case

Figure 1.5: Lexical environment manipulation

We now describe how this mechanism interacts with the lexical environment. Figure 1.4 shows an example of a JavaScript lexical environment, the lexical environment order being represented by vertical arrows. To access variable `x` in the current scope l_0, it is first searched in l_0 itself and its prototype chain. As `x` is not found, the lexical environment is followed and the variable is looked up in the declarative environment record above. Only `y` is defined here and the search continues in l_1 and its prototype chain. This time, `x` is found in location l_2, thus the value returned is 2. Note that the value 1 of `x` present in l_g is shadowed by more local bindings, as well as the value 3 present in l_3.

Some special objects have a particular use. We have already encountered the global object, located at l_g. This object is where global variables are stored. As most objects, the special location l_{op} is present in its prototype chain[2], which we describe below. The global object is always at the top of the lexical environment. A second special object is the prototype of all objects, `Object.prototype`, located at l_{op}. Every newly created object has a field $@proto$ bound to either l_{op} or an already declared object. It has some functions which thus can be called on every object (but they can be hidden by local declarations) such as `toString` or `valueOf` (see Section 1.2.4). Finally, the prototype l_{fp} of all functions, `Function.prototype`, is a special object equipped with function-specific methods.

Finally, the JavaScript execution context carries a special location, which can be accessed through the keyword **`this`**. We name it the **`this`**-location. It is generally bound either to l_g, or to a specific object from which the current function has been called. For instance, if a function `f` is called as a method of an object `o`, as in `o.f ()`, then the **`this`** location will be bound to `o` during the execution of `f`. However, if we call it through a local variable, as in the following code, then **`this`** will be bound to l_g.

```
var x = o.f ;
x ()
```

[2] This is actually implementation dependent, but let us state that is it the case for the sake of readability.

1.2.3.2 Manipulating Lexical Environments

We have seen how accesses are performed in a heap. Let us see how the lexical environment is changed over the execution of a program, and in particular over the execution of a function call, a `with` statement, a `new` expression, or an assignment. A graphical representation of those changes is summed up in Figure 1.5. In this figure, the orange blocks represent newly allocated locations or declarative environment records.

As usual in functional programming languages, the current lexical environment is saved when defining functions. When calling a function, the saved lexical environment is restored, adding a new scope at the front of the chain to hold local variables and the arguments of the call. The special location `this` can also be updated. The "`with (o){...}`" statement puts the object `o` in front of the current lexical environment to run the associated block. In the "`new f (...)`" case, the function `f` is called with the `this` assigned to a new object. The implicit prototype @*proto* of this new object is copied from the `prototype` field of `f`. This is how prototype chains are usually created. The newly created object, which may have been modified during the call to `f` by "`this.x = ...`" statements, is then returned. There are some corner cases with different behaviours, but we do not detail them.

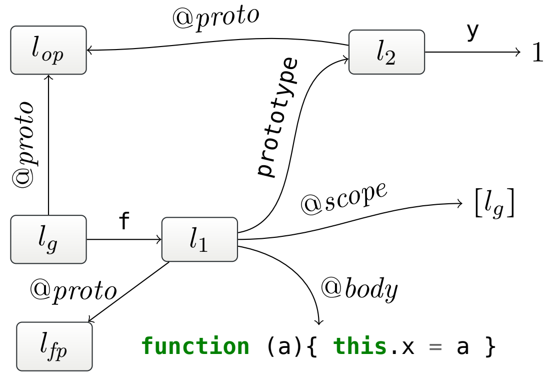
Example. A heap modified by a `new` operator at Line 7 of the program of Figure 1.6a is shown in Figure 1.6c. Upon executing the `new` instruction, the function `f` is called with `this` pointing to a new object l_3. The function body is executed, adding an `x` field to `this`. This object located at l_3 is then returned, setting its implicit prototype @*proto* to the value of the field `prototype` of `f`.

Targeted assignment, of the form `e.x = 4`, are straightforward: the expression `e` is computed and should return a location l (technically, it is converted into a location, see Section 1.2.4 for more information). Then the field `x` is written with value `3` in the object at location l. For untargeted assignments, such as `x = 4`, things are more complex. The first scope for which the searched field is defined is selected in the current lexical environment, following the same variable look-up rules as above. The variable is then written in this scope. If no such scope is found, then a new variable is created in the global scope.

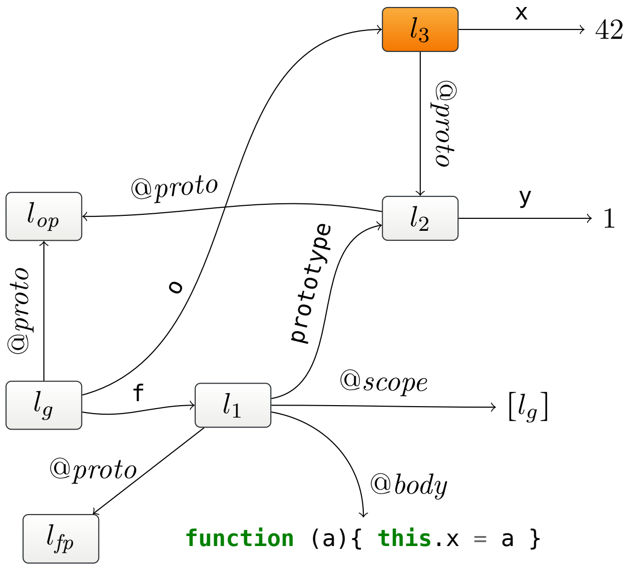
Figure 1.7 describes the effect of the assignment `x = 4`. Location l_1 is the first to define `x` in its prototype chain (in l_2). The new value of `x` is then written in l_1. Note that it is not written in l_2, allowing other objects which have l_2 in their prototype chain to retain their old value for `x`. Nevertheless, if one accesses `x` in the current lexical environment, the new value 4 is returned. This approach may lead to surprising behaviours, as we now illustrate. Note that in a JavaScript program which does not use the `with`-construct, the only object environment record of the lexical environment is l_g; this greatly simplifies the situation, getting back to the usual scoping of variables.

```
1 var f = function (a){
2     this.x = a
3 }
4
5 f.prototype = {y : 1} ;
6
7 var o = new f (42)
```

(a) The executed program

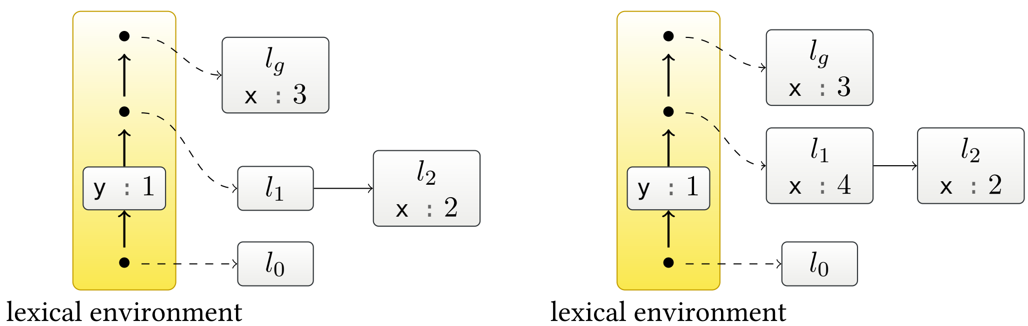


(b) Heap's state at Line 6



(c) Heap's state at the end of Line 7

Figure 1.6: Effect of the `new`-construct



(a) Before **(b)** After

Figure 1.7: Effect of an assignment

Example. Consider Program 1.1. If it is executed in an empty heap, it returns 42. Indeed, when defining `f`, no such function already exists, and `f` is thus stored in the global scope. When `f` accesses `a` upon its call, the object `o` is in the lexical environment (as the call is executed in the lexical environment of the definition of `f`), thus the result is 42. Now, consider the same program in a slightly different scope, where `Object.prototype.f` has been set to a function (say f' = `function (){ return 18 }`). The code `var o = {a : 42}` is almost equivalent to `var o = new Object () ; o.a = 42`, the object `o` (at location l_o) has thus an implicit prototype set to `Object.prototype` (which is l_{op} by definition). Figure 1.8 shows a representation of the heap at Line 3. As there is a variable `f` defined in the lexical environment at position l_o (because of its prototype chain to l_{op}), the assignment is local to l_o and not global. At Line 5, the variable look-up for `f` returns the function f', which is found in the prototype of the global object l_g (at this point the lexical environment only contains l_g), and not the function `f` defined in the `with` block. Thus the call executes the function f', which returns 18.

There are many other subtleties in JavaScript's semantics which can be used to execute arbitrary code. Getters, setters, as well as field attributes (which can disable some actions to be performed on fields—for instance preventing their removal) also feature a lot of semantic peculiarities. For the context of this thesis, however, I consider implicit type conversions to be a better example of JavaScript's unexpected complexity.

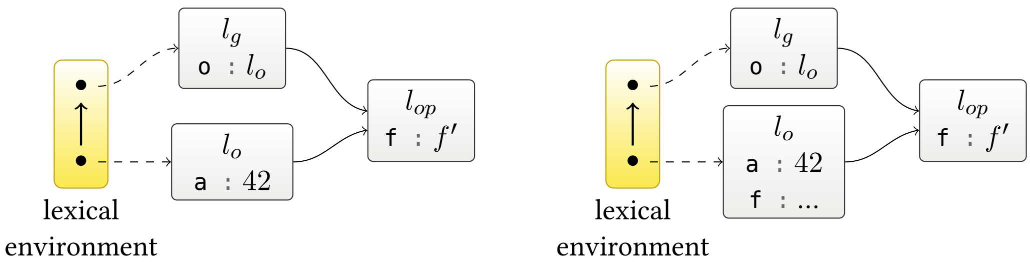
1.2.4 Implicit Type Conversions

As said in Section 1.2.2, there are only six types of values in JavaScript. Arrays, functions and other high level constructs are considered to be special kinds of objects; they each have a specific prototype corresponding to their kind, which provides them with some default attributes and methods. Some of these special objects differ in some aspects from user-defined objects, usually by having special internal fields which are not always limited to hold JavaScript values. For instance, functions have some internal fields storing their inner program as well as their definition scope. The specification also makes some exceptions about the behaviour of some fields of these special kinds of objects. For instance, the field `length` of arrays is automatically updated if we write a new field in the array, even if we did not use a special setter for this.

To illustrate how type conversion works, I will use the surprising result [Kle13] that, in the default environment, every JavaScript string can be computed by an expression only composed of the six characters `(`, `)`, `[`, `]`, `+`, and `!`. For instance, Program 1.2 is an encoding of my last name "Bodin" in JavaScript. Indentation has been added for readability. This section is meant to be a gentle introduction on how complex the JavaScript semantics can be, and in particular how the execution of a seemingly benign program can yield the execution of various unexpected parts of the ECMAScript standard.

```
1 var o = { a : 42 } ;
2 with (o) {
3     f = function (){ return a }
4 } ;
5 f ()
```

Program 1.1: One of the pitfalls of the `with`-construct



(a) At the beginning of Line 3 **(b)** At the end of Line 3

Figure 1.8: Heap state of Program 1.1

Let us consider Program 1.2 step by step. The simplest type conversion is from a value to a boolean. It typically happens when a value stands as the condition of an `if` construct, a loop, or before calling some boolean operators such as the boolean negation `!`. The conversion to boolean, called *ToBoolean* in the ECMAScript specification, returns `true`, except for the values `NaN`, `0`, the empty string “”, `false`, `null`, and `undefined`. The operation *ToBoolean* is called an *abstract operation* in the specification: it is used to define the semantics of JavaScript but can not be directly accessed or changed in a JavaScript heap. This allows us to define the two booleans in our character restriction. A new empty array can be created using `[]`, which is almost equivalent to `new Array ()`. It is an object, and thus coerces to the boolean `true`. We build `false` using `![]` and `true` using `!![]`.

The *ToPrimitive* abstract operation is much more dangerous: it converts its argument to a primitive value—often a number or a string. If the argument is an object, then two of its methods may be called: `toString` and `valueof`. This is dangerous as these can then execute arbitrary code: implicitly converting an object to a primitive value in an unknown environment can thus yield arbitrary side-effects. Also note that `toString` is just a JavaScript function—there is no guarantee that it will actually return a string.

The conversion to a number, triggered by the unary operator `+`, calls the abstract operation *ToPrimitive*. If it terminates, an operation is performed on the result: string are parsed (returning `NaN` if the string does not parse as a number), `true` is converted to `1`, `null` and `false` to `0`, and `undefined` to `NaN`. We can thus build 0 with `+![]` and 1 with `+!![]`.

```
((+![]+(![])[
        ([][        ([]+![])[+[]]
                    +([![]]+[][[]])[+!![]+[+[]]]
                    +([]+![])[!![]+!![]]
                    +([]+!![])[+[]]
                    +([]+!![])[!![]+!![]+!![]]
                    +([]+!![])[+!![]]
            ]+[])[!![]+!![]+!![]]
        +(!![]+[][      ([]+![])[+[]]
                        +([![]]+[][[]])[+!![]+[+[]]]
                        +([]+![])[!![]+!![]]
                        +([]+!![])[+[]]
                        +([]+!![])[!![]+!![]+!![]]
                        +([]+!![])[+!![]]
            ])[+!![]+[+[]]]
        +([]+[][[]])[+!![]]
        +([]+![])[!![]+!![]+!![]]
        +([]+!![])[+[]]
        +([]+!![])[+!![]]
        +([]+[][[]])[+[]]
        +([][        ([]+![])[+[]]
                    +([![]]+[][[]])[+!![]+[+[]]]
                    +([]+![])[!![]+!![]]
                    +([]+!![])[+[]]
                    +([]+!![])[!![]+!![]+!![]]
                    +([]+!![])[+!![]]
            ]+[])[!![]+!![]+!![]]
        +([]+!![])[+[]]
        +(!![]+[][      ([]+![])[+[]]
                        +([![]]+[][[]])[+!![]+[+[]]]
                        +([]+![])[!![]+!![]]
                        +([]+!![])[+[]]
                        +([]+!![])[!![]+!![]+!![]]
                        +([]+!![])[+!![]]
            ])[+!![]+[+[]]]
        +([]+!![])[+!![]]
    ])[+!![]+[+[]]]
+(!![]+[][      ([]+![])[+[]]
                +([![]]+[][[]])[+!![]+[+[]]]
                +([]+![])[!![]+!![]]
                +([]+!![])[+[]]
                +([]+!![])[!![]+!![]+!![]]
                +([]+!![])[+!![]]
    ])[+!![]+[+[]]]
+([]+[][[]])[!![]+!![]]
+([![]]+[][[]])[+!![]+[+[]]]
+([]+[][[]])[+!![]])
```

Program 1.2: A program equivalent to `"Bodin"` in the default environment

```
counter = 0 ;

console.log (+[18]) ;   // Prints 18.

Array.prototype.toString = function (){
    counter++;          // Performs a side-effect.
    return 42
} ;

console.log (counter) ;  // Prints 0.
console.log (+[18]) ;    // Prints 42.
console.log (counter)    // Prints 1.
```

Program 1.3: A program with potentially unexpected implicit type conversions

The abstract operation *ToString* performs a similar operation to convert a value into a string: it calls *ToPrimitive*, then converts the result (**`true`** returns the string “`true`”, etc.). Surprisingly, calling *ToString* on **`false`** returns the string “`false`”, whose conversion to a boolean, through *ToBoolean*, is **`true`**. The object `Array.prototype` has a method `toString` predefined in the default environment: it calls the *ToString* abstract operation on all the values in the indexes of the array, then concatenates their results separated by the character `,` (this is a simplification of what really happens, but it is enough to understand what follows). For instance, `+[18]`, converts `[18]` to a primitive, resulting in the string “`18`”; this string will then be converted to a number, resulting in the numeric value 18. This is only the result we would get in the default environment: if we change the conversion function for arrays as in Program 1.3, the result may be different.

Addition in JavaScript can be used for both string concatenation and numerical addition. The `+` operator is treated in two steps: first, *ToPrimitive* is applied on both its arguments. If one of them results in a string, then the other is converted into a string and the string concatenation is performed. Otherwise, both are converted into numbers and the numerical addition is performed. This double meaning of the JavaScript binary operator `+` can lead to surprising behaviours. In our example, as the empty array `[]` converts (in the default environment) to the empty string, adding the empty array to a value converts this value to a string. Thus `[] + ![]` results in the string “`false`”. We can also build numbers by adding booleans (which will be converted to numbers): `!![] + !![]` results in 2.

The conversion of a value to an object may allocate a new object, but shall not result in other side effects. It is usually performed when accessing a field of a non-object value. For instance `"str".length` first converts `"str"` into an object: it is almost equivalent to `(new String ("str")).length`. The difference between the former and the latter program is that the latter will first perform a variable look-up on the global variable `String`, which can be redefined, whilst the former will always choose the same object, referenced as `String` in the initial heap. The resulting object has for instance a field `length` set to `3`, as well as three fields `0`, `1`, and `2`, respectively set to the strings “`s`”, “`t`”, and “`r`”.

To build our first letters, we have to understand the construct `e1[e2]`. This construct evaluates the two expressions `e1` and `e2` to the values v_1 and v_2. It then converts v_1 onto an object l, and v_2 onto a string str. Finally it performs a field look up in the prototype chain of l, looking for the field str. If the field is not found in the prototype chain, then the value **`undefined`** is returned. The expression `e.f` presented in the previous section is just a syntactic sugar for `e["f"]`.

Now, consider how the letter `n` in Program 1.2 is built, on line 47: `([] + [][[]]) [+!![]]`. To evaluate `[][[]]`, we build an empty array and try to access one of its fields. The name of this field is given by the conversion to a string of `[]`, which is the empty string. As there is no field indexed by the empty string in the prototype chain of the empty array, the value **`undefined`** is returned. This value is then added to the empty array, converting it to the string "`undefined`". We then access its field indexed by `+!![]`, which results in `1`. We thus convert the string "`undefined`" to an object and access its field `1`. This returns the string "`n`". The letters `d` and `i` are build similarly.

The letter `o` is a little more complex as it comes from the conversion into a string of a function, which should be parsable as a function (but no guarantee is given on the behaviour of this printed function), thus starting by "`function`". We thus have to get any function and add it to the empty array. In this case, the chosen function is `[].filter`: the empty array has a prototype pointing the `Array` object, which contains a variety of fields—including `filter`. This method is accessed from Line 38 to 43, then Line 44 extracts the letter `o` out of this string. The letter `B` is built from **`false`**`.constructor`, which is a function named "`Boolean`". Converting it into a string returns a string starting by "`function Boolean`" from which we can extract `B`. We assumes here that the interpreter puts exactly one space between "`function`" and "`Boolean`". This is not guaranteed by the specification, but most interpreters do it like this[3]. The word "`constructor`" is built from Line 2 to 36.

The complete program builds the string "`Bodin`". This however requires the program to be executed in the default environment: if we change some important conversion functions before running the program, we can get totally different results. For instance, if we first run Program 1.3, then run Program 1.2 in the new environment, we will get **`NaN`** as a result instead. We will also have the variable `counter` set to 86 at the end of the execution.

The purpose of this section was not to teach how to obfuscate JavaScript programs, but to show that type conversions can appear in many places if we are not paying attention to them. Notice how these type conversions are arbitrary: they do not follow from the memory model, for instance. This will have some consequences on the size of the formalisation of Chapter 2 and on how we choose to tackle the problem of analysing JavaScript in Chapter 4. We conclude this section by explaining the two equalities operators in JavaScript. The double-equal comparison `==` performs an algorithms which will convert both its arguments in order to compare them. For instance, comparing **`true`** `== "1"` converts

[3] Purists might prefer to call `Object.prototype.toString` on a boolean, as the position of the `B` is here specified. Unfortunately, reaching this function requires much more space and is not shown here.

both arguments into numbers and thus yields `true` (but `true == "true"` yields `false`). In particular, == can lead to arbitrary side effects. To avoid surprises, it is thus good practise to use the triple-equal comparison ===, which can not yield side effect; it also behaves as can be expected, without performing any type conversion. In a more general comment, any implicit type conversion in a program can already be a source of security breach: an analyser focussing on these constructs is thus already interesting.

1.2.5 The `eval` Construction

JavaScript's most fearful feature is the `eval` construct. It behaves as a function which takes a string as a parameter, then runs this string as if being a JavaScript program. This dynamic features is how JavaScript implements reflection. Other constructs show similar reflectiveness—this includes the function `Function`, which is used in most JavaScript programs. The reflectiveness of JavaScript can be frightening, but consider that in the context of a browser, reflection is unavoidable: a JavaScript program can add a new `script` node to the current webpage through its Document Object Model (DOM), the browser will then execute the corresponding JavaScript code anyway.

There are cases where the usage of `eval` is useful—typically for code loading. Adding a new `script` node into the current DOM would work, but there are no[4] way to know when the new script is executed, or what is the current stage of download. Instead, a small JavaScript program can be used to load a set of JavaScript files, possibly to decompress them. When ready, this program can then evaluate each of the loaded files.

These reflective constructs are very complex for program analyses: analyses fail when encountering an `eval` whose string is unknown. There have been some studies [Ric+10; Ric+11] about how `eval` is used in practise: it is used to load libraries, but not only. Most of these uses can however be rewritten to an `eval`-free program. One common usage of `eval` happens when exchanging information with servers. Instead of writing a parser, the information transfer between the servers and the program can be performed under JavaScript's object syntax; it can then be evaluated directly using `eval` to get the corresponding object, which is directly usable. An alternative is to use JSon, a parser for this syntax available in JavaScript's default environment. Richards et al. [Ric+11] state that 83% of the uses of `eval` in real-world programs can be replaced by an `eval`-free program.

1.2.6 Standard Libraries

ECMAScript defines an initial environment where the global object already contains a lot of predefined objects and functions, such as `Array` or `parseInt`. They behave as if part of a default library loaded at the beginning of each execution. Among these objects and functions, some could be programmed directly in JavaScript alone, and some contain additional special features.

[4] As far as I know, there is currently no standard documenting such a feature.

1. Let O be the result of calling *ToObject* passing the `this` value as the argument.
2. Let *lenVal* be the result of calling the *Get* internal method of O with argument "length".
3. Let n be *ToUint32* (*lenVal*).
4. Let *items* be an internal List whose elements are, in left to right order, the arguments that were passed to this function invocation.
5. Repeat, while *items* is not empty
   a) Remove the first element from *items* and let E be the value of the element.
   b) Call the *Put* internal method of O with arguments *ToString* (n), E, and `true`.
   c) Increase n by 1.
6. Call the *Put* internal method of O with arguments "length", n, and `true`.
7. Return n.

Program 1.4: The specification of `Array.prototype.push`

```
Array.prototype.push = function (){
    var O = new Object (this) ;
    var lenVal = O.length ;
    var n = lenVal >>> 0 ;
    for (var i = 0; i < arguments.length; i++){
        var E = arguments[i] ;
        O[n] = E ;
        n++
    }
    O.length = n ;
    return n
}
```

Program 1.5: A possible implementation of `Array.prototype.push`

Most of the behaviours of the special fields of `Array` can be expressed only using the part of JavaScript presented above. There thus exists a minimal subset of JavaScript features which is enough to define the complete initial environment. Such a subset is usually named *core-JavaScript*. However, in this dissertation, I will name the core of JavaScript the part about very basic features, such as functions, arrays, object look-up, and implicit conversions, but without most of their initially defined attributes; this corresponds to Chapters 1 to 14 of the ECMAScript 5 specification. Chapter 15 of ECMAScript describes the initial environment; it can be seen as the standard library of JavaScript, with many features to manipulate arrays, strings, and other constructs. Section 2.4.2 provides more details about how the ECMAScript is organised.

For instance, the function `Array.prototype.push` writes its argument in a new field of the `this`-location, whose name depends on the field already defined in this object. Program 1.4 shows its specification in ECMAScript 5 and Program 1.5 a possible implementation[5]: it is implementable in JavaScript. On the other hand, the field `length` of an array in Java-

[5] All the constructs of this program have already been detailed in this chapter expect the >>> construct: it is an unsigned shift, here used to convert the `length` attribute to a number representing an integer.

Script is difficult to define using only the core: it is a numerical value greater than any number n such that the field $\mathit{ToString}\,(n)$ of the accessed array is defined and deletable. It is technically definable using getters and setters as there are only a finite number of possible array indexes, but it would not be practical for an interpreter (even a toy one) to enumerate 2^{32} fields at each access. The `length` field of arrays depends on the structure of its argument as a whole, and we consider it to be a special feature.

1.2.7 Parsing

Usually, parsing is not a very difficult part when interpreting a programming language. It also is not a problem for analysers in general, as they can reuse the same parsers used by real interpreters. But in order to analyse constructs such as `eval` when the input string is only partially known, analysers have to precisely catch the syntax of some constructs. In the case of JavaScript, this is particularly challenging. To illustrate this, let me reuse an example from wtfjs [LLT10]. The two JavaScript Programs 1.6a and 1.6b only differ by one semicolon, and yet are parsed very differently, as indicated by the syntax highlighting. Semicolons in JavaScript are not mandatory, as stated by the introduction of Section 7.9 of the ECMAScript specification about automatic semicolon insertion:

> Certain ECMAScript statements (empty statement, [...], **`return`** statement, and **`throw`** statement) must be terminated with semicolons. Such semicolons may always appear explicitly in the source text. For convenience, however, such semicolons may be omitted from the source text in certain situations. These situations are described by saying that semicolons are automatically inserted into the source code token stream in those situations.
> [...] When, as the program is parsed from left to right, a token (called the offending token) is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if [the offending token stands on a new line].

In other words, semicolons are added at the beginning of every line which did not parse correctly. The elided parts of this quotation describe some exceptions; such as Program 1.7a which is parsed as Program 1.7b and throws the value **`undefined`**, although **`throw`** `a + b` is a valid JavaScript program.

Let us focus back on the example of Programs 1.6a and 1.6b. In the case with semicolon, the first line is an assignment. The second line is then parsed as a regular expression matching a sequence of `1`s followed by the string "`"//`". We then confront it to the string "`1"//`" (see Section 1.2.4 for more details), which matches. The second expression thus returns **`true`**. Program 1.6c rewrites this program in a readable form. Let us now focus on Program 1.6b. The automatic semicolon insertion does not fire as the code makes sense: the character `/` is interpreted as a division symbol. We thus get one divided by one multiplied by a string, which returns **`NaN`**. The last two `/` characters are interpreted as the beginning of a single line comment: we get a program equivalent to Program 1.6d.

```
n = 1 ;
/1*"\/\//.test (n + '"//')
```

(a) With semicolon

```
n = 1
/1*"\/\//.test (n + '"//')
```

(b) Without semicolon

```
n = 1 ;
(new RegExp ('1*"//')).test (n + '"//')
```

(c) How Program 1.6a is parsed

```
n = 1 / 1 * "///.test (n + '"
```

(d) How Program 1.6b is parsed

Program 1.6: Two variants of a program which yield different results

```
throw
    a + b
```

(a) The source code of a program

```
throw
    ; a + b
```

(b) How it is parsed

Program 1.7: A common parsing pitfall

Removing one `;` character (which is usually automatically inserted) changed the interpretation of all the following characters of the source code. In particular, it is not possible to know whether a given part of a program is commented without knowing the whole context. In practise, this does not hinder the construction of analysers if we are ready to use an external parser from widely used JavaScript engines. It can hinder the analysis of a program using `eval`; but being imprecise on such programs is acceptable, as their behaviours are already very complex. I will avoid to play with JavaScript's syntax in this dissertation to keep things readable.

1.2.8 Strict Mode

The strict mode is the most important feature added from version 3 to 5 of ECMAScript. It is an official variant of the JavaScript language designed to have lexical scoping, among other nice properties. This variant of the language is switched on by the flag `"use strict"` at the beginning of a program or a function's code. In practise, the semantics changes are indicated along the specification by steps such as the following one.

1. If *code* is strict mode code, then let *strict* be **`true`** else let *strict* be **`false`**.

Program 1.8 shows an example of a program without lexical scoping (in non-strict mode). The variable `x` of Line 5 can reference three different `x`s: the field `x` of the object of Line 4, the argument `x` of the function `f` declared Line 3 and called Line 10, and the global variable `x` declared Line 1. Because of the **`with`** construct, it initially refers to the field `x` of Line 4: Figure 1.9 shows the program's state when Line 5 is executed for the first time. The heap is modified along the execution by the construct **`delete`** of Line 6: it deletes the first appearance of the variable `x` in the environment, following the look-up rules presented in Section 1.2.3.1, returning **`true`** if the field was successfully deleted and **`false`** otherwise. The global variable `x` of Line 1 is in this case not deletable: calling `f` prints `with` and `argument`. The variable `x` of Line 5 thus refers to three different places of the heap.

Strict mode aims at preventing such difficulties. Running Program 1.8 in strict mode would be rejected as the two constructs **`with`** and **`delete`** `x` break scopes: the strict mode defensively rejects any construct preventing to statically determine where variables stand in the heap. Strict mode also prevents the **`this`** construct to accidentally take the value of the global object. These restrictions help increasing both the efficiency of JavaScript interpreters and the security of JavaScript programs, as much more information can be statically extracted from a given program.

1.3 Implementation Dependent Features

The specified JavaScript does not provide any way of interacting with the external environment, called *host environment* in ECMAScript. The reason is that JavaScript can appear in several places, which are not all browsers. For instance, Node.js runs in terminals and can not interact with any webpage: specifying webpage interactions in ECMAScript would thus be meaningless for Node.js. To this end, the specification allows implementations to add some features in the initial JavaScript global object. These features can have any effect, on the JavaScript heap or the host environment, as explained by the following extract of the ECMAScript specification.

> [It] is expected that the computational environment of an ECMAScript program will provide not only the objects and other facilities described in this specification but also certain environment-specific host objects, whose description and behaviour are beyond the scope of this specification except to indicate that they may provide certain properties that can be accessed and certain functions that can be called from an ECMAScript program.

Examples of implementation dependent features include interactions with webpages (by changing their DOM), servers, files, or with the user. It can also be features of future versions of JavaScript which are already available in the interpreter; such as the unstandardised or experimental features in Mozilla Firefox [05].

These additional features are important for analysers: by accessing an unspecified field of an initially-defined object, we can not be sure that it is not present. Consider for instance Program 1.1, which relies on the fact that l_{op} does not have a field named `f`: it is not a safe program as it assumes that the initial state does not contain some fields—which is nevertheless accepted by ECMAScript. In the analyses defined during this thesis, I thus chose domains such as the one presented in Figure 4.3: such domains are able to state that

```
1  var x = "out" ;
2
3  function f(x){
4      with ({ x : "with" }){
5          do console.log (x)
6          while (delete x)
7      }
8  }
9
10 f ("argument")  // prints "with", then "argument"
```

Program 1.8: A JavaScript program without lexical scoping

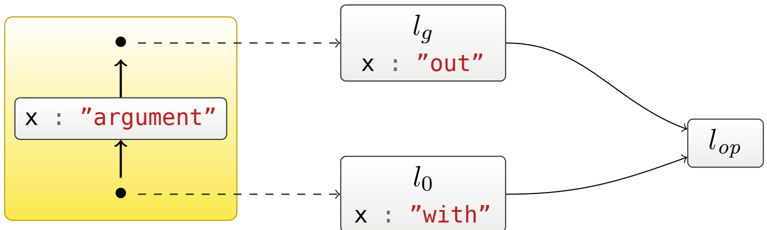


Figure 1.9: State of Program 1.8 at the first execution of Line 5

we do not know whether a given variable is defined or not. In particular, in order to safely analyse a JavaScript program, every unspecified field of an object should be initially set to express this absence of knowledge.

1.4 Conclusion

We have seen in this section that JavaScript is a successful programming language. In particular, its success has led to the emergence of JavaScript programs manipulating sensitive data. We have also seen that JavaScript have been designed to be flexible, enabling to easily mix different JavaScript programs from various sources in the same environment. Analysing JavaScript programs is thus important for security reasons.

We have also seen how complex JavaScript is. The language can be extremely versatile, yielding side effects where they are the least expected. The language is complex enough to make the results of an analyser be considered dubious if this analyser is not accompanied by some sort of certification. There are many ways to certify programs; common instances include testing, or making an expert of JavaScript closely rereading the source code of the analyser. This thesis focusses on formal methods, and in particular on formally proven techniques: analysers are accompanied by a mathematical proof of their soundness. In particular, the next chapter aims at providing a formal model of JavaScript.

2 Formalising JavaScript

Bah, je sais pas, on a des boulots super, tout ça... mais parfois je me sens comment dire... un peu détaché du monde réel, tu vois ?

Yörgl, by Gilles Roussel (pen named Boulet) [Rou09]

Because they can interfere with sensitive data, there are cases in which JavaScript programs need to be analysed. But for a language such as JavaScript, trusting analysers can be difficult. Intensive testing can help increase the trust, but semantic exceptions can be difficult to catch. This happened for instance for ADSafe [Cro08]—a program checking language restrictions to ensure the sandboxing of external JavaScript programs—which was shown flawed [Pol+11b] at the time (the flaw is now fixed).

Proof assistants such as Coq [CHP+84] or Isabelle/Hol [NPW02] have proved to be very powerful tools to trust programs. The CompCert project [Ler+08] has for instance been able to build and certify an optimising compiler for C. This compiler is proven to be free of compilation bugs, leading to safer programs in critical software.

We would thus like to certify a JavaScript analyser to be correct. But correct with respect to what? On one hand, the official ECMAScript semantics is written in prose and not directly usable for our means. On the other hand, JavaScript interpreters are far too complex to be basis of correctness proofs. We are thus in the need of a JavaScript formal specification. This led to the JSCert [Bod+12] project, whose primary goal is to provide a formal semantics for JavaScript in Coq. It involved 8 persons for a year. During this project, we produced from the 200 pages of the ECMAScript standard about 20,000 lines of Coq code, including 4,000 for the JSCert specification (see Section 2.5), 3,000 for a reference interpreter named JSRef (see Section 2.6), and 4,000 for the proof of its correctness (see Section 2.7). This chapter is mostly based on the JSCert formalisation [Bod+14]. It aims at showing how JSCert has been defined, and why it can be trusted.

2.1 Language Specifications

A language specification is a way to precisely describe what are the programs considered correct and how a given program executes. It can come in various forms:

- an implementation of a compiler or an interpreter (as for PHP),
- a document in prose with varying degrees of rigour (for instance, the C standard [Gro11] and ECMAScript 5 are fairly precise and complete),

$$\frac{e_1 \rightarrow e_1'}{@\, e_1\, e_2 \rightarrow @\, e_1'\, e_2}\ \textsc{eval-func} \qquad \frac{e_2 \rightarrow e_2'}{@\, v_1\, e_2 \rightarrow @\, v_1\, e_2'}\ \textsc{eval-arg} \qquad \frac{}{@\ (\lambda x.\ e_1)\ v_2 \rightarrow e_1\,[x/v_2]}\ \textsc{app}$$

(a) In small-step

$$\frac{}{v \Downarrow v}\ \textsc{exec-val} \qquad \frac{e_1 \Downarrow \lambda x.\ e_3 \qquad e_2 \Downarrow v_2 \qquad e_3\,[x/v_2] \Downarrow v_3}{@\, e_1\, e_2 \Downarrow v_3}\ \textsc{exec-app}$$

(b) In big-step

$$\frac{}{v \Downarrow v}\ \textsc{Val} \qquad \frac{e_1 \Downarrow v_1 \qquad @_1\, v_1\, e_2 \Downarrow v_3}{@\, e_1\, e_2 \Downarrow v_3}\ \textsc{app-func} \qquad \frac{e_2 \Downarrow v_2 \qquad @_2\, v_1\, v_2 \Downarrow v_3}{@_1\, v_1\, e_2 \Downarrow v_3}\ \textsc{app-arg} \qquad \frac{e_1\,[x/v_2] \Downarrow v_3}{@_2\ (\lambda x.\ e_1)\ v_2 \Downarrow v_3}\ \textsc{app-beta}$$

(c) In pretty-big-step

Figure 2.1: The rules of the λ-calculus in different semantics styles

- a formal specification (as for Standard ML [MTM97]), which can be paper-based or mechanised (that is, computer-based, using proof assistants).

Programmers usually use unspecified features and thus rely on implementations—this however varies along language communities. On the other hand, compilers, interpreters, and analysers usually rely on prose documents or on formal specifications. There are thus several specifications for a given programming language, each usage associated to one of them. These different definitions do not necessarily agree with each other. Each form of specification comes with advantages and drawbacks. The main advantage of having a reference interpreter is the interaction with the language community. It can however lead to overspecifications, which can be controlled in formal frameworks.

Through its history (see Section 1.1.1), JavaScript started to be specified by implementations, then to have a prose specification: ECMAScript 1. Formal specifications—including JSCert— then appeared (although they are not official). JavaScript is also equipped with some test suites [ECM10; Moz13] but they do not cover the full language yet. It is important to keep this diversity in mind: an interpreter or an analyser specified by one specification of JavaScript is not automatically correct with respect to another.

2.1.1 Formal Specifications

Formal specifications are mathematical objects describing the behaviours of programs. They take several forms, for instance denotational or axiomatic semantics. We shall focus on operational semantics, which are defined by a transition system. They define a predicate $\Downarrow$ relating a program p and its input σ (or *semantic context*) to a result r. The input can be program inputs, but also the initial environment. Similarly, the result can be just a value or a whole environment. Returning an environment is a way to describe side-effects. The behaviours of a semantics are the triples $\sigma, p \Downarrow r$ recognised by the semantics. If a semantics displays more (respectively less) behaviours than another—for the inclusion order—, then it is complete (respectively correct) with respect to this other semantics. 2

Operational semantics can take several forms. Figure 2.1 shows the rules of the λ-calculus (in call-by-value) in several kinds of semantics, presented below. The predicate $\Downarrow$ is instantiated as follows: there is no semantic context σ in pure λ-calculus, e stands for expressions, and v for values. The set of values only consists of λ-expressions, which are considered modulo α-conversion. We write $e\,[x/v]$ for the capture-avoiding substitution of all free occurrences of x in e to v. The application is written $@$.

A *small-step semantics* [PK87], or *rewriting semantics*, is a semantics focussing on transitions. Such semantics introduce intermediary terms in which the computation has been partially performed. For instance, Rule EVAL-FUNC of Figure 2.1a performs an internal computation in a term e_1; the resulting terms $@\,e_1'\,e_2$ is not fully computed. Furthermore, the computation of e_1 may have resulted in an error: in this regard, the term $@\,e_1'\,e_2$ is a special term—an intermediary term. Focussing on transitions enables us to define interleaved programs: if two programs are executed in parallel, each of them can make a step without altering the other thread. This has been useful for reasoning about parallel computations and is one of the reasons why small-step semantics are common nowadays.

A *big-step semantics* [Kah87], or *natural semantics*, focusses on the results of programs. It is natural to reason about a program specified in big-step style by reasoning directly over the structure of program derivations. Intermediary terms are no longer needed, hidden in the structure of derivations. However, some behaviours are difficult to fit in this formalism.

There exist variants of these two main types of semantics. In this dissertation, we will be interested in *pretty-big-step* [Cha13], a restriction of big-step semantics to the following constraints. First, rules can not refer to inductive premises (that is, to $\Downarrow$) more than twice. Second rules can not refer to future computations: for instance Rule EXEC-APP of Figure 2.1b (in big-step) does not respect this constraint, as it requires the term e_1 to evaluate to a λ-abstraction $\lambda x.\, e$. This constraint forces rules to be local. The only way to know whether Rule EXEC-APP applies is to evaluate e_1 and check that the result has the requested form. The evaluation of e_1 can be arbitrarily complex, or may not terminate: it is impossible to locally know whether the rule applies. In pretty-big-step (see Figure 2.1c), we first evaluate e_1 in a separate rule, and only then we have to decide whether the next rule applies: only local knowledge is required to know whether a rule applies. To this respect, pretty-big-step is closer to small-step than big-step. Section 4.3 provides more detailed information about pretty-big-step and its formalisations. The restrictions of pretty-big-step fits the description of languages with flow-breaking instructions, in particular, it avoids rule-duplication. Section 2.5.2.1 elaborates more on this subject.

These semantic styles can be translated from one to the other. Pretty-big-step semantics are already a subset of big-step semantics. Ciobâcă [Cio13] proposed a compilation from small-step semantics to big-step semantics, later refined to a compilation from small-step to pretty-big-step [PM14].

2.1.2 Specifications using Coq

Coq is a proof assistant based on the calculus of inductive constructions [CH88]. It can be seen as a purely functional language with rich types. It includes syntactic restrictions on how fixed points can be defined to ensure that every function terminates. This makes Coq functions similar to mathematical functions. Defining partial functions is still possible by defining an inductive type `option A` parametrised by the type `A`, as shown below[1].

```
Inductive option (A : Type) : Type :=
  | Some : A → option A
  | None : option A.
```

More generally, Coq implements generalised data structures (GADT) [XCC03]. This enables to define derivation trees directly as a Coq data structure. Program 2.1 shows such a derivation definition on the rules of Figure 2.1b. The type `var` is the set of program variables and is left as a parameter. The substitution is defined as a fixed point. It terminates because the inductive type `expr` only contains finite structures. The **`ifb`** construct is described in Section 3.4.1: it is used to test whether the two variables `x` and `x'` of Program 2.1b are identical. For the sake of simplicity, we do not consider the case where `ex` has `x'` as a free variable in Line 7 of Program 2.1b. Instead, we refer the interested reader to other Coq formalisations of the λ-calculus [Ter95]. The derivation is described in Program 2.1c as a tree structure of type `expr → expr →` **`Prop`**. The **`Prop`** type is a special construct denoting propositions; We shall not extend on this subject here: the reader can consider that defining an element of type `derivation e v` amounts to prove that `e` $\Downarrow$ `v`.

The type `derivation e v` depends on the terms `e` and `v`, Coq allows types to depend on terms—in other words, Coq is dependently typed. This enables us to express very precise properties about programs. The Coq framework also enables to define fixed points whose return value lives in **`Prop`**. This amounts to prove properties by induction over some term or derivation. This parallel between terms and proofs is called the Curry-Howard isomorphism [CF58; How80]. Defining proofs terms is a complex task. Proofs are usually defined using tactics, that is, programs which help building proof terms. The unproven premises of tactic proofs are sent back to the user, making proof assistants interactive. A detailed presentation of a proof using tactics is shown in Sections 2.7.3 and 5.3. Apart from these particular proofs, proofs are not detailed in this dissertation.

[1] This type is actually already included in the standard Coq library.

```
Variable var : Type.

Inductive expr : Type :=
  | variable : var → expr
  | lambda : var → expr → expr
  | app : expr → expr → expr.
```

(a) Syntax

```
Fixpoint substitute e x ex :=
  match e with
  | variable x' =>
      ifb x = x' then ex else e
  | lambda x' e' =>
      ifb x = x' then e
      else lambda x' (substitute e' x ex)
  | app e1 e2 =>
      app (substitute e1 x ex) (substitute e2 x ex)
  end.
```

(b) Variable substitution

```
Inductive derivation : expr → expr → Prop :=
  | exec_val : forall x e,
      derivation (lambda x e) (lambda x e)
  | exec_app : forall e1 e2 x e3 v2 v3,
      derivation e1 (lambda x e3) →
      derivation e2 v2 →
      derivation (substitute e3 x v2) v3 →
      derivation (app e1 e2) v3.
```

(c) Semantics

Program 2.1: Specifying the semantics of Figure 2.1b in Coq

Symmetrically to inductive definitions, Coq also accepts coinductive definitions [RL09]. Whilst inductive definitions are structurally finite (at least for the intuition: carried functions can make things complex), coinductive structures can be infinite. These constructs can be seen as lazy: they are only computed when being pattern matched, and the computation will stop when enough information has been computed for the pattern matching. Section 4.4.2 uses coinduction to define potentially infinite derivation trees.

2.2 Large Scale Formalisations

The JSCert project was not the first to provide a formal semantics of a complex programming language. This section presents some related works in the domain of formal specification of programming languages in general and JavaScript in particular. JSCert shares many of the challenges faced by these works.

2.2.1 For Languages Other Than JavaScript

One of the most prominent, fully formalised, presentations of a programming language is Standard ML [MTM97]. A mechanised specification [LCH07] was later given in the Twelf theorem prover [PS99]. Unlike Standard ML, few programming languages are designed with formalism in mind. This raises a considerable challenge to mechanisation.

There have been a lot of efforts on mechanised language specifications in Isabelle/Hol. For instance Norrish [Nor98] specified a small-step semantics of C and used it to prove simple programs. He also proved some invariants of the semantics, for instance that undefined behaviours get propagated. However, his semantics has not been related to implementations. Another example is about transmission control protocols (TCP) [Bis+06], in which the authors specified TCP from several implementations. The specification was validated by several thousand test traces captured from implementations.

In the context of the CompCert project [Ler+08], Blazy and Leroy [BL09] built a verified optimising compiler for CompCert C—a subset of C—as well as a Coq proof that the generated code behaves as one of the possible behaviours of the source program. The CompCert project initiated major technological breakthroughs in Coq mechanisation, some of which has been used in JSCert. The semantics of C chosen for the CompCert compiler has been related to the specification of C in prose. Several projects are based on CompCert. CompCertTSO [Šev+11] adapts CompCert for the x86 weak memory model [Alg+10]. Besson et al. [BBW15] extends CompCert to give a semantics to more programs. The language specification of CompCert has then been used as a basis of certified analysers [App11; Jou+15]: these analysers have been defined and certified in Coq.

Proof assistants require some effort to get used to. Researchers are beginning to explore how to make mechanised specification easier. The $\mathbb{K}$ framework is designed specifically for writing and analysing language definitions using small-step [ER12]. In particular, Roşu and Şerbănuţă [RŞ10] define an executable formal semantics of C in $\mathbb{K}$. This formalisation has been tested against the GCC test suite [Fre10]. Besides being executable, the semantics also comes with an explicit-state model checker. In contrary to CompCert, this semantics is related to the C compilers through tests. The relatively recent $\mathbb{K}$ framework has received considerable interest from various other authors. An instance is a formalisation of PHP in $\mathbb{K}$ by Filaretti and Maffeis [FM14]. Similarly to $\mathbb{K}$, Ott [Sew+10] is another framework designed to specify semantics; it provides a domain-specific tool to define programming languages. The Ott framework is able to automatically translate to Isabelle/Hol and Coq. Owens [Owe08] defines a mechanised semantics of Caml Light using Ott.

There are many more examples of mechanised specifications of programming languages. For instance, the work of Syme [Sym99] for an Isabelle/Hol version of the formal Java semantics of Drossopoulou and Eisenbach [DE97], Gurevich's work [Gur94] for an executable formalisation of the C# standard, and Farzan et al.'s work [Far+04] for an executable formal semantics of the version 1.4 of Java in small-step. The formal semantics [Bat+11; BDG13] of the concurrency of C++, which has a real impact on the C11 standard [Gro11].

2.2.2 Formal JavaScript Specifications

JSCert is not the first formalisation of JavaScript. Notably, ECMAScript 1, the first standard of JavaScript, was based on several implementations (mainly Netscape's and Microsoft's). This section aims at presenting the most important formalisations.

The firsts to propose a formal type system for a subset of JavaScript were Anderson et al. [AGD05] and Thiemann [Thi05] in 2005. To prove type-soundness, they formalised idealised cores of the language which abstracted away features not crucial for their type analyses. Since then, researchers have studied various typed JavaScript subsets and static analyses [PSC09; GL09; JC09; CHJ12; JMT09; Chu+09; HS12; PLR11; Gua+11; PLR12]. For example, Thiemann [JMT09] used abstract interpretation to develop a tool inferring abstract types for the full language, although the formal theory only works for a subset. Others have studied information flow [Chu+09], with some [HS12] proving their results in Coq. All these techniques have been helpful for addressing specific safety problems. None provide general-purpose analyses, most do not work with the full language and those who prove soundness do so with respect to their abstract models rather than the ECMAScript semantics or an actual concrete implementation. The security issues identified in some works [MT09b; MMT09; MMT10] demonstrate that the semantic subtleties of corner cases of the language crucially matter. Moreover, empirical analysis [Ric+11] confirms that some of the language features which are usually ignored by researchers are important for actual web programmers.

The first formal semantics of JavaScript to be executable was the one of Herman and Flanagan [HF07]. The authors presented an interpreter of a non-trivial subset of ECMAScript 4 written in Standard ML, which is in turn formalised [Mil+97]. Having an executable semantics enables testing. It also has the advantage of being easily read by functional programmers. The drawbacks are a loose correspondence with the specification and implementation details which sometimes obscure the semantics of the language features. Because of JavaScript's non-local features, they had to use Standard ML's exceptions—these features are directly modelled using pretty-big-step in JSCert.

The first full semantics for ECMAScript 3 was defined by Maffeis et al. [MMT08]. It covers the whole language—apart from a few corner cases, such as regular expressions, dates, and machine arithmetic. The (large) formalisation has been done in small-step style and proves some theorems about determinacy and well-definedness of the language. This work has been useful to prove the soundness of security-related JavaScript subsets [MT09b; MMT09; MMT10], and influenced the definition of further JavaScript form-

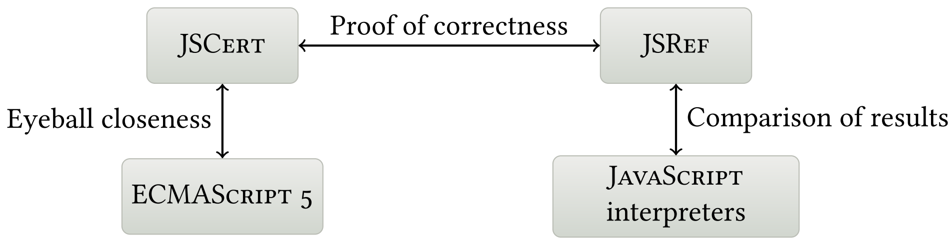


Figure 2.2: How JSCert is related to JavaScript

alisations, such as Secure ECMAScript [Tal+11], and the big-step semantics of core JavaScript [GMS12; BDM13]. However, this work was not mechanised, which makes building further works on top of it or proving language properties difficult.

Guha et al. [GSK10] proposed a different approach to develop language semantics. They provide a translation (named *desugaring*) from JavaScript to an executable language called λ_{JS}, based on a λ-calculus with references. Their aim was to develop provably sound type systems to reason about the safety of client-side web applications. They targeted the implementation of ECMAScript 3 in Mozilla Firefox. The semantics was validated by testing it against JavaScript test suites [ECM10; Moz13]. The λ_{JS} semantics has then been extended to the strict mode of ECMAScript 5 [Pol+12a] under the name S5. An unpublished, small-scale Coq formalisation of λ_{JS} has been announced on the Brown PLT blog [Pol+12b]. It features a Coq model of λ_{JS}, as well as some properties about the λ_{JS} language, such as progress and some invariants.

The work on λ_{JS} has been influential in proving properties of well-behaved JavaScript typed subsets, where programmers accept restrictions on full JavaScript in exchange for safety guarantees. For example, Politz et al. [Pol+11a] present a type system for λ_{JS} which captures the informal restrictions enforced by ADSafe. Fournet et al. [Fou+13] define a translation between F*—a subset of Microsoft F# with refinement types—and λ_{JS}. The authors show that their encoding is fully abstract, implying that the safety properties respected by a source F* program are preserved when translated to JavaScript and run on a trusted web page. The λ_{JS} formalisation has since been related to JSCert by Materzok [Mat16], who provides a Coq formalisation of λ_{JS}—both in the form of an operational semantics and an interpreter, both representations being proved correct and complete with respect to each other. Materzok then proved that the desugaring of λ_{JS} is correct with respect to JSCert, effectively relating both formalisations.

Since JSCert came out, a new formal specification of JavaScript has been published: KJS [PSR15] is a $\mathbb{K}$ specification if JavaScript. It has the advantage of being runnable, whilst being rule-based (in small-step). Section 2.10 discusses more about it.

To conclude these sections of related work, there are two main ways to relate a formal semantics to a language: one can test that the semantics agrees on the result of tests suites, or one can relate the semantics to the official specification of the language. In other words, a formal semantics can be trusted either because it produces the same results than interpreters, or because it is close to its prose specification. JavaScript has the chance of having both: the main specification is written in prose, but a lot of test suites exist—in particular those used to relate λ_{JS} to JavaScript. To get the full trust of a formal specification of such a subtle language as JavaScript, we need both ways. The JSCert project is thus split into two parts: a specification—named JSCert—, and an interpreter—named JSRef.

2.3 Methodology

Given the size of JavaScript's semantics, a formal semantics of JavaScript is likely to have bugs. As this semantics is meant to be the basis of further works, it has to be trusted. But how to trust such a semantics? There are two ways on which a semantics can be related to JavaScript: by relating it to JavaScript's prose specification, or by relating it to test suites. The idea behind JSCert is that, to trust a formal specification, it has to be certified by both certification methods: to the prose specification and to the test suites.

To be the closest possible to ECMAScript, JSCert has to use the same data structures, even if it means losing efficiency or readability. This closeness enables us to point where each part of ECMAScript is represented in JSCert and conversely. If someone disagrees on an interpretation of the prose specification, it is easy to locate which part of JSCert should be changed—this also applies if someone wants to formalise a variant of JavaScript, such as the one executed in a real-world interpreter. On the other hand, to be able to run JSCert on test suites, it is accompanied by a JavaScript interpreter named JSRef. JSRef is proven correct with respect to JSCert. Figure 2.2 sums up the relations between JSCert and JavaScript. Section 2.7 explains how the trust is established.

JSCert is not the first formal or executable semantics of JavaScript, but it is the first semantics for the entire core language, closely reflecting the official standard, both executable and formalised in a proof assistant. Reflecting the structure of the specification has several advantages over a translational approach (such as λ_{JS}): the JavaScript programmer intuition is better reflected, and the semantics is robust to local changes[2].

2.4 The ECMAScript Standard

ECMAScript 5 has not been defined with formal specification in mind. It has not been optimised for conciseness, reuse, or readability. It also contains a lot of copy/pasted parts. A semantics of this size has to be read with doubt, which is why JSCert was being needed. We already encountered an extract of the ECMAScript standard in Program 1.4: it is presented as an algorithm written in pseudo-code, each step being executed in order[3]; it uses structures useful for interpreters, but not necessarily for reasoning.

[2] Unfortunately, the changes between ECMAScript 5 and ECMAScript 6 are not local. This makes JSCert stuck for now in ECMAScript 5. But there are some promising workaround presented in Section 2.8.

[3] This is the reason why this dissertation categorises extracts of ECMAScript as programs.

Interestingly, the ECMAScript standard is precise and non-ambiguous, with two notable exceptions. We have already discussed one in Section 1.3: the standard allows the interpreter to add objects in the initial environment, whose behaviours are completely free. Second, the specification of the `for-in` construct defines very loosely the enumeration order of field names; this is discussed in more details in Section 2.9.

Completion triples are a good example of the non-proof-friendly structures of ECMAScript; they are the result of the evaluation of an expression, a statement, or a program. Completion triples carry information about executed flow-breaking instructions, such as `throw`. They are composed of a type, an optional value, and an optional label. The given type is one of *Normal*, *Return*, *Break*, *Continue*, or *Throw*: *Normal* indicates that the result comes from a non-flow-breaking instruction. Any completion whose type is not *Normal* is called an *abrupt completion*. The optional value describes the result of an expression or a statement (if the type is *Normal*), the value carried by a return statement (if the type is *Return*), or the object describing the exception being thrown (if the type is *Throw*). When this value is absent, we shall speak of an empty or an undefined value (not the value `undefined`). The optional label is only used for the *Break* and *Continue* types, it is used to divert the execution flow to specific target labels.

As functional programmers, we (the JSCert team) started JSCert by defining completion triples as in Program 2.2a, assuming that some invariants hold in the specification—for instance that a completion triple with type *Break* would not carry a value, as `break` statements do not return such completion triples. However, there are places in the specification breaking this assumption, starting with the sequence of statements whose specification is shown in Program 2.3. As can be seen Step 6, the resulting completions triple s defined Step 3 has been updated into a completion triple with a new value V, defined Step 5. As a consequence, the statement `break l` returns the completion triple $(Break, empty, \mathtt{l})$, but the sequence `1 ; break l` returns $(Break, 1, \mathtt{l})$. The assumed invariants thus do not hold. This made us change the definition of completion triples to be closer to ECMAScript, by removing our initial constraints. The new definition is shown in Program 2.2b.

Given the complexity of JavaScript's semantics, changing definitions can yield a lot of effort to rewrite the rules. It is thus important to use the same structures as the specification, even where the specification definitions do not fit intuition or hinder readability. The intuition of researchers can be different from the one of the ECMAScript committee. For instance, the loop unrolling property does not hold in ECMAScript 5—that is `while (e) s` is not equivalent to `if (e){s ; while(e) s}`—the `while`-construct mixing completion triples differently than the sequence [Theb] (see next section). This has been fixed in ECMAScript 6, though. A structure based on the assumption that some invariants always hold may broke in future versions of ECMAScript. As ECMAScript nowadays changes quickly, being the closest possible to the original specification is critical to keep up the pace. In addition, using the same structure is a good way to increase the trust of correspondence between the two specifications, as discussed in details in Section 2.7.

```
Inductive res :=
  | res_normal : option value → res
  | res_break : option label → res
  | res_continue : option label → res
  | res_return : value → res
  | res_throw : value → res.
```

(a) Completion triples as they were defined in an early version of JSCert

```
Inductive restype := (* result type *)
  | restype_normal
  | restype_break
  | restype_continue
  | restype_return
  | restype_throw.

Inductive resvalue := (* result value *)
  | resvalue_empty : resvalue
  | resvalue_value : value → resvalue
  | resvalue_ref : ref → resvalue.

Inductive reslabel := (* result label *)
  | reslabel_empty : reslabel
  | reslabel_string : string → reslabel.

Record res := { (* completion triple *)
  res_type : restype ;
  res_value : resvalue ;
  res_label : reslabel }.
```

(b) Current JSCert completion triples

Program 2.2: JSCert completion triples

"`s1 ; s2`" is evaluated as follows.

1. Let sl be the result of evaluating `s1`.
2. If sl is an abrupt completion, return sl.
3. Let s be the result of evaluating `s2`.
4. If an exception was thrown, return $(\mathit{Throw}, V, \mathit{empty})$ where V is the exception. (Execution now proceeds as if no exception were thrown.)
5. If $s.value$ is empty, let $V = sl.value$, otherwise let $V = s.value$.
6. Return $(s.type, V, s.target)$.

Program 2.3: Semantics of the sequence of statements in ECMAScript 5

2.4.1 Running Example: the `while` Statement

We have seen many interesting features in the semantics of JavaScript in Chapter 1, including prototype-based inheritance in Section 1.2.3 and implicit type conversions (with potential side effects) in Section 1.2.4. All of these features are properly described by the JSCert semantics. We focus here on the `while` statement, as it is simple enough to show how JSCert has been built, whilst showing interesting aspects of the formalisation.

Program 2.5 shows the prose specification of the `while` construct as it appears in the ECMAScript 5 standard. Its specification is relatively short in comparison to other constructs such as `switch`, whose specification spreads on more than one page. The pseudo-code of ECMAScript is similar to usual imperative programming language—in particular, it leaves completely implicit two major aspects of its semantics. The first aspect is the threading of the mutable state: ECMAScript 5 assumes that there is one global heap storing objects, and that the instructions in the pseudo-code can modify this heap. The second aspect is the propagation of aborting states through expressions: aborting states (such as exceptions) occurring in statements are explicitly propagated (as in Step 4 of Program 2.3) but this is not the case in expressions, where propagation is left implicit. The reason of this difference of treatment may be that, in contrary to expressions, statements usually update the value of returned completion triples, such as in Steps 4 and 6 of Program 2.3. This issue being a potential source of ambiguities, it has been solved in ECMAScript 6, in which the propagation of aborting states is always explicitly specified.

Let us describe the pseudo-code of Program 2.5 in details. The basic structure is standard: repeat the loop body until the loop condition evaluates to `false`, or until the body of the loop produces an abrupt completion, such as a `break`, a `return`, or an exception. Consider Step 2b: the result of the expression `e` is not necessarily ready to be used, as it can be a *reference* to an object field. The internal *GetValue* function is used to dereference it. In addition, JavaScript uses the internal function *ToBoolean* to implicitly coerce the loop guard to a boolean before attempting to test it (see Section 1.2.4 for more details). Internal functions such as *GetValue* and *ToBoolean* can not abort, which justifies the usage of the functional notation $ToBoolean\left(GetValue\left(exprRef\right)\right)$. Now, consider Step 2e. JavaScript enables labelled `break` and `continue` statements to refer to an outer loop: the "current label set" refers to the set of labels which are associated with the current loop (a loop may be associated to several labels). For instance, in the program `outer: while (1){ inner: while (1){ break outer } }`, the result of the `break` statement gets propagated through the inner loop as if it were an exception.

Note that `while` loops have a return value V. The returned value of statements can be accessed using the `eval` construct. As detailed in Step 5 of Program 2.3, the output value of a sequence of statements is the last value produced by a statement in their body. This leads to the results shown in Program 2.4: there is no inner computation Line 1, and the value V is empty all along, resulting in the value `undefined`. In Line 4, the inner loop computes

```
1  eval ("out: while (true){ while (true){ break out }}") ;
2      // Returns undefined.
3
4  eval ("out: while (true){ while (true){ 'in' ; break out }}") ;
5      // Returns "in".
6
7  eval ("out: while (true){ 'middle' ; while (true){ 'in' ; break out }}") ;
8      // Returns "in".
9
10 eval ("out: while (true){ 'middle' ; while (true){ break out }}") ;
11     // Returns "middle".
12
13 eval ("out: while (true){ 'middle' ; while (true){ undefined ; break out } }")
14     // Returns undefined.
```

Program 2.4: Return values of various `while` statements

"`while (e) s`" is evaluated as follows.

1. Let V = *empty*.
2. Repeat
   a) Let $exprRef$ be the result of evaluating `e`.
   b) If $ToBoolean\,(GetValue\,(exprRef))$ is `false`, return (*Normal*, V, *empty*).
   c) Let $stmt$ be the result of evaluating `s`.
   d) If $stmt.value$ is not empty, let $V = stmt.value$.
   e) If $stmt.type$ is not *Continue* or $stmt.target$ is not in the current label set, then
      i. If $stmt.type$ is *Break* and $stmt.target$ is in the current label set, then
         A. Return (*Normal*, V, *empty*).
      ii. If $stmt$ is an abrupt completion, return $stmt$.

Program 2.5: Semantics of the `while` construct in ECMAScript 5

the inner value `"in"`, which is propagated. In Line 7, both the inner and outer loop creates a result, the last result being propagated. Line 13 shows that the value `undefined` behaves exactly the same as any other value, overwriting the previous result `"middle"`.

2.4.2 What JSCert does Not Specify

The ECMAScript 5 standard is a document of 16 chapters, with more than 200 pages; it largely consists of pseudo-code in the style of Programs 2.3 and 2.5, with some prose clarifications. The standard is separated into the following chapters:

- Chapters 1 to 4, as well as Chapter 16, (9 pages in total) describe how the standard itself should be read.
- Chapters 5 to 7 (21 pages) describe how JavaScript programs should be parsed. The grammar of each construct is given in the next chapters simultaneously to their specifications: these three chapters explain specifically how to use the grammar to build an abstract syntax tree (AST).

- Chapters 8 and 9 (23 pages) describe the values manipulated in JavaScript, as well as some internal functions to manipulate them, such as *ToBoolean*.
- Chapter 10 (12 pages) describes the context in which a JavaScript program is executed (see Sections 1.2.3 and 2.5.1.2 for more details).
- Chapters 11 to 14 (40 pages) describe how each construct should be executed (expressions, statements, and programs). Programs 2.3 and 2.5 come from this part.
- Chapter 15 (104 pages) covers the native library: JavaScript comes with a build-in library to manipulate its structures, such as arrays and strings (see Section 1.2.6). This chapter also includes the definition of objects such as `Math` and `Date`.

JSCert provides a specification of the main part of the language: the syntax (as an AST); the semantics of expressions, statements and programs; and most native library functions exposing internal features of JavaScript—in particular the methods of the objects `Object`, `Function`, and `Error`. However, most of Chapter 15 have not been formalised. The objects `Array`, `String`, and `Date` involve hundreds of methods. Furthermore, most of these constructs do not interact with any internal feature of JavaScript. As seen in Section 1.2.6, these functions could be implemented as plain JavaScript code: see Section 2.8 for a discussion about it. The **`for-in`** construct has not been formalised because the standard defines it very loosely; this is discussed in more details in Section 2.9.

JSCert does not specify the parsing of JavaScript programs. This is notable as JavaScript enables reflection (see Section 1.2.5). Furthermore, as seen in Section 1.2.7, parsing JavaScript is unusually complex: building a certified parser of JavaScript is a difficult task. JSRef uses the Esprima parser [Hid12], a heavily tested JavaScript parser.

To conclude, JSCert provides a specification of Chapters 8 to 14 of the ECMAScript specification. The rest can either be completed using an external parser, or using features directly implemented in JavaScript's core language (as discussed in Section 2.8).

2.5 JSCert: JavaScript Specification in Coq

The formal development in Coq of JSCert [Bod16] consists of five main parts, shown in Figure 2.3 with the Coq files implementing each part and their dependencies. The first part describes the syntax and data structures—such as heaps and scopes—which are used to describe the state of a JavaScript programs; both JSCert and JSRef share these definitions. The second part contains a collection of auxiliary definitions, such as functions used to convert primitive values to booleans or strings. These first two parts are described in Section 2.5.1. The next two parts correspond to JSCert (Section 2.5.2) and JSRef (Section 2.6). The last part consists of the correctness proof, proving that any result computed by JSRef is correct with respect to the semantics from JSCert (Section 2.7).

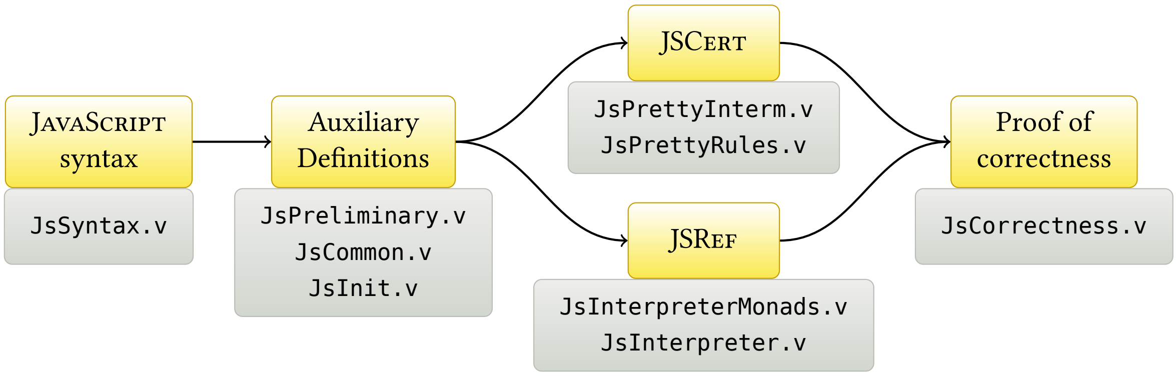


Figure 2.3: General structure of JSCert and JSRef, with the corresponding Coq files

2.5.1 Syntax and Auxiliary Definitions

JSCert and JSRef share the same data structures, from the grammar of JavaScript to the representation of the memory. This section describes these structures.

2.5.1.1 Abstract Syntax Tree

Parsing is not modelled in JSCert, which directly manipulates an AST. In the JSRef interpreter, the AST is obtained by running Esprima extended with some interface code.

The grammar of JavaScript expressions and statements is shown (in part) in Program 2.6. Compared to the JSCert specification, this inductive is very short. Indeed, the complexity of JavaScript resides its corner cases, not in its constructs. A JavaScript program consists of a list of function definitions and statements, as well as a strictness flag (see Section 1.2.8). The body of a function definition is itself a JavaScript program. The argument of a call to `eval`, once parsed, is also a JavaScript program.

2.5.1.2 Execution State

As explained in Section 1.2.3, a JavaScript program is always executed in a given *state* and in a given *execution context*. The state is a structure used to make side-effects global: it is propagated over every side-effect construct. The execution context is used to associate variables to their values. The state consists of two heaps: the object heap (which we will often simply call "the heap") and the heap of environment records. The object heap is represented as a finite map from locations to objects. We have seen in Section 1.2.3 that objects have some special fields in JavaScript, such as their prototype. In total, the record representing objects contains 25 components; this includes the *field map*—which maps every normal fields to their values—, as well as some optional fields, such as the body and scope for functions. These special fields enable to tag objects with different behaviours; for instance, arrays have a special *Delete* internal method. Not all of these special fields carry a normal value; for instance, the body of a function carries a program.

```
Inductive expr := (* expressions *)
  | expr_this : expr
  | expr_identifier : string → expr
  | expr_literal : literal → expr
  | expr_object : list (propname * propbody) → expr
  | expr_function : option string → list string → prog → expr
  | expr_access : expr → expr → expr
  | expr_call : expr → list expr → expr
  | expr_binary_op : expr → binary_op → expr → expr
  | expr_assign : expr → option binary_op → expr → expr
  (* ... *)

with stat := (* statements *)
  | stat_expr : expr → stat
  | stat_block : list stat → stat
  | stat_var_decl : list (string * option expr) → stat
  | stat_if : expr → stat → option stat → stat
  | stat_while : label_set → expr → stat → stat
  | stat_with : expr → stat → stat
  | stat_throw : expr → stat
  | stat_return : option expr → stat
  (* ... *)

with prog := (* programs *)
  | prog_intro : strictness_flag → list element → prog

with element := (* program elements *)
  | element_stat : stat → element
  | element_func_decl : string  → list string → prog → element.

with propbody := (* items in object initializers *)
  | propbody_val : expr → propbody
  | propbody_get : prog → propbody
  | propbody_set : list string → prog → propbody

(* ... *).
```

Program 2.6: A snippet of JSCert AST

The field map binds field names to *field attributes*, rather than directly to values. Indeed, JavaScript enables to tag some fields with special properties such as writable or enumerable. These tags are then used in the semantics to associated them with special actions. There are two kinds of field attributes, corresponding to the two ways an object field can be defined. *Data field attributes* store a value and whether the value is writable, enumerable, or configurable. *Data accessor attributes* store two values (a getter and a setter) as well as two tags, enumerable and configurable.

ECMAScript 5 suggests that field attributes should be represented as a record with six optional fields. In particular, the specification of the function *DefineOwnProperty* involves the construction of a field attribute which explicitly manipulates records using arbitrary subsets of the six optional fields. However, in many other places, the standard uses exactly four fields, implicitly making the assumption that the considered field attribute is either a data field attribute or a data accessor attribute, depending from the context. In order to strictly follow the standard, JSCert provides two distinct representations of data fields: the first consists of a record with six optional fields; whilst the second consists of an inductive type with two cases, one for data field attributes and one for data accessor attributes, both represented as records with exactly four mandatory fields. The overhead of defining conversion functions between the two forms was negligible compared to the benefits of avoiding the pollution of many rules with accesses to optional fields.

In addition to the object heap, a state also contains a heap of environment records. As described in Section 1.2.3.1, there are two kinds of environment records: declarative environment records provide the local scoping of function calls, and object environment records point to an object in the object heap. Environment records are stored in a data structure similar to the object heap. The outermost environment of a lexical environment is always an object environment record pointing to the global object. Function objects store a lexical environment in one of their internal fields (called $@scope$ in Figure 1.6).

2.5.1.3 Execution context

The execution context indicates how to interpret JavaScript programs. It contains a flag indicating whether the current execution is in strict mode (see Section 1.2.8). It also stores the `this` value and the lexical environment (see Section 1.2.3.1).

2.5.2 JSCert

The semantics of JavaScript statements is given in JSCert by a judgement of the form $S, C, t \Downarrow_s o$, where t denotes a statement, S denotes the state (see Section 2.5.1.2), C denotes the execution context (see Section 2.5.1.3), and o denotes the *output*. There are similar judgements $\Downarrow_e$, $\Downarrow_i$, and $\Downarrow_p$ for expressions, internal reductions, and programs. The output is a pair of the final state and the completion triple produced by the evaluation (see Section 2.4); it is represented by the Coq type `out`. The judgement $\Downarrow_i$ for internal reductions has a return type different from the three others, as it can result in many different types—for instance, some return field descriptors, which can not be represented in the type `out`. They are thus associated with a more general return type, named `specret T` for "special return", which is parametrised by the type `T` of returned values. To add further complexity, internal reductions may call arbitrary user code which may terminate with an abrupt termination, such as throwing an exception; their return type is thus not uniform: it returns a modified state and a term of type `T` when the computation is successful, but returns a term of type `out` otherwise. This behaviour is captured by the `specret` type shown below; it features two constructors: `specret_val` for when the requested type `T` is successfully built, and `specret_out` for when an exception is thrown.

```
Inductive specret T :=
  | specret_val : state → T → specret T
  | specret_out : out → specret T.
```

2.5.2.1 Pretty-big-step Semantics Style in JSCert

The ECMAScript standard uses a very specific pseudo-code, as we can see in Program 2.5: it is basically a sequence of steps of the form "Let r be the result of evaluating t", with some additional branching constructs, such as "If" and "Repeat". Such let-steps directly relate a term t to its result r, as in big-step semantics style. However, each step of ECMAScript can abort, breaking the control flow. In big-step style, this would duplicate rules, resulting in a semantics difficult to match with the standard. We translate below Program 2.5 in big-step style, from Step 2a to Step 2c. The first rule considers the case in which `e` aborts in Step 2a, the second when it evaluates to a value converted into the boolean **`false`**, and the third when it evaluates to a value converted into **`true`**, but the statement `s` aborts. Previous computations are repeated in each latter rule. If a construct is described by n steps in the standard, it would be translated in big-step into approximately n rules with up to n premises. The big-step style thus does not scale to the complexity of JavaScript.

$$\frac{S, C, \mathtt{e} \Downarrow_e o \qquad \textbf{abort}\, o}{S, C, V, \mathtt{while\ (e)\ s} \Downarrow_s o}$$

$$\frac{S, C, \mathtt{e} \Downarrow_e \mathtt{vret}\ S'\ v \qquad S', C, \mathit{ToBoolean}\,(\mathit{GetValue}\,(v)) \Downarrow_i \mathtt{false}}{S, C, V, \mathtt{while\ (e)\ s} \Downarrow_s \mathtt{vret}\ S'\ V}$$

$$\frac{\begin{array}{c} S, C, \mathtt{e} \Downarrow_e \mathtt{vret}\ S'\ v \\ S', C, \mathit{ToBoolean}\,(\mathit{GetValue}\,(v)) \Downarrow_i \mathtt{true} \qquad S, C, \mathtt{s} \Downarrow_s o \qquad \textbf{abort}\, o \end{array}}{S, C, V, \mathtt{while\ (e)\ s} \Downarrow_s \mathtt{vret}\ S'\ V}$$

Big-step semantics are not very common nowadays because of concurrency, which is better expressed in small-step style. This issue is not a problem for JavaScript, as no concurrency is defined in the standard: a JavaScript program blocks its host. In particular, a looping JavaScript program in a webpage freezes its browser (or browser-tab). This behaviour enables us to consider other types of semantics styles, such as pretty-big-step.

2

The pretty-big-step semantics style [Cha13] discussed in Section 2.1.1 appeared to be a good match for JSCert: similar to big-step, it directly relates terms with results, but it also avoids rule duplication. In this set-up, each step of the standard is associated to an intermediary term performing one local computation; after this local computation is performed, the computation either goes to the next step (represented by an intermediary term), or stops in case of abortion. This allows JSCert and ECMAScript to be close to each other, thus increasing the trust expected from JSCert. Figure 2.4 shows the JSCert rules corresponding to Program 2.5 describing the `while` construct; it is written `stat_while L e t` in Coq, where `e` is the guard, `t` the body, and `L` is a set of labels (used to manage `break` and `continue` statements), as shown in Program 2.6. We now show the close correspondence between the steps of Program 2.5 and the JSCert rules.

Step 1 of the ECMAScript specification says “Let $V = \mathit{empty}$”; in JSCert, Rule red-stat-while redirects the computation of `stat_while L e t` into the intermediary term `stat_while_1 L e t resvalue_empty`. This intermediary term carries all the information of the original `while` construct, with the additional information that the value of V is `resvalue_empty`, which is the representation in JSCert of an empty value. Step 2 consists of the loop; in JSCert, we may loop back to this point at any time using the `stat_while_1` intermediary term, as if it were the label of a Goto instruction.

Now consider Steps 2a and 2b; these steps represent a common pattern in ECMAScript: first, we evaluate some sub-expression, then we perform a *GetValue* and a type conversion (here as a boolean) on the result. Note how much is left implicit in ECMAScript: the expression evaluation and the type conversion could diverge or abort[4]; and both the expression evaluation and the type conversion could have side effects on the program state. This pattern occurs so frequently that we introduced a special intermediate form to handle it, while making these side effects, divergence, and abortion propagation clear: in Rule red-stat-while-1, the intermediate form `spec_expr_get_value_conv` takes care of the evaluation of `e`, its *GetValue*, and its type conversion. We specify which type to convert to using the term `spec_to_boolean`.

The remaining work of Step 2b is performed by Rule red-stat-while-2-false. As the type conversion may have side effects, Rule red-stat-while-2-false takes its initial state S from the result of the type conversion as given by the intermediary term, ignoring the other one given (written _ in Figure 2.4). The new state can not be sent directly to this rule as it would require to match in Rule red-stat-while-1 over the output `y` of the type

[4]This is actually not the case for boolean conversions, but it may happen in string or number conversions.

RED-STAT-WHILE

$$\frac{S, C, \texttt{stat\_while\_1 L e t resvalue\_empty} \Downarrow_s o}{S, C, \texttt{stat\_while L e t} \Downarrow_s o}$$

RED-STAT-WHILE-1

$$\frac{\begin{array}{c} S, C, \texttt{spec\_expr\_get\_value\_conv spec\_to\_boolean e} \Downarrow_i \texttt{y} \\ S, C, \texttt{stat\_while\_2 L e t rv y} \Downarrow_s o \end{array}}{S, C, \texttt{stat\_while\_1 L e t rv} \Downarrow_s o}$$

RED-STAT-WHILE-2-FALSE

$$\frac{}{\_, C, \texttt{stat\_while\_2 L e t rv (vret } S \texttt{ false)} \Downarrow_s \texttt{out\_ter } S \texttt{ rv}}$$

RED-STAT-WHILE-2-TRUE

$$\frac{S, C, \texttt{t} \Downarrow_s o_1 \qquad S, C, \texttt{stat\_while\_3 L e t rv } o_1 \Downarrow_s o}{\_, C, \texttt{stat\_while\_2 L e t rv (vret } S \texttt{ true)} \Downarrow_s o}$$

RED-STAT-WHILE-3

$$\frac{\begin{array}{c} \texttt{rv}' = \begin{cases} \texttt{res\_value R} & \text{if } \texttt{res\_value R} \neq \texttt{resvalue\_empty}, \\ \texttt{rv} & \text{otherwise} \end{cases} \\ S, C, \texttt{stat\_while\_4 L e t rv' R} \Downarrow_s o \end{array}}{\_, C, \texttt{stat\_while\_3 L e t rv (out\_ter } S \texttt{ R)} \Downarrow_s o}$$

RED-STAT-WHILE-4-CONTINUE

$$\frac{\begin{array}{c} \texttt{res\_type R} = \texttt{restype\_continue} \wedge \texttt{res\_label\_in R L} \\ S, C, \texttt{stat\_while\_1 L e t rv} \Downarrow_s o \end{array}}{S, C, \texttt{stat\_while\_4 L e t rv R} \Downarrow_s o}$$

RED-STAT-WHILE-4-NOT-CONTINUE

$$\frac{\begin{array}{c} \neg(\texttt{res\_type R} = \texttt{restype\_continue} \wedge \texttt{res\_label\_in R L}) \\ S, C, \texttt{stat\_while\_5 L e t rv R} \Downarrow_s o \end{array}}{S, C, \texttt{stat\_while\_4 L e t rv R} \Downarrow_s o}$$

RED-STAT-WHILE-5-BREAK

$$\frac{\texttt{res\_type R} = \texttt{restype\_break} \wedge \texttt{res\_label\_in R L}}{S, C, \texttt{stat\_while\_5 L e t rv R} \Downarrow_s \texttt{out\_ter } S \texttt{ rv}}$$

RED-STAT-WHILE-5-NOT-BREAK

$$\frac{\begin{array}{c} \neg(\texttt{res\_type R} = \texttt{restype\_break} \wedge \texttt{res\_label\_in R L}) \\ S, C, \texttt{stat\_while\_6 L e t rv R} \Downarrow_s o \end{array}}{S, C, \texttt{stat\_while\_5 L e t rv R} \Downarrow_s o}$$

RED-STAT-WHILE-6-NORMAL

$$\frac{\texttt{res\_type R} \neq \texttt{restype\_normal}}{S, C, \texttt{stat\_while\_6 L e t rv R} \Downarrow_s \texttt{out\_ter } S \texttt{ R}}$$

RED-STAT-WHILE-6-ABORT

$$\frac{\texttt{res\_type R} = \texttt{restype\_normal} \qquad S, C, \texttt{stat\_while\_1 L e t rv} \Downarrow_s o}{S, C, \texttt{stat\_while\_6 L e t rv R} \Downarrow_s o}$$

Figure 2.4: JSCert semantics of while loops

RED-EXPR-ABORT

$$\frac{\texttt{out\_of\_ext\_expr e} = \texttt{Some } o \qquad \textbf{abort}\, o \qquad \neg\texttt{abort\_intercepted\_expr e}}{S, C, \texttt{e} \Downarrow_e o}$$

RED-STAT-ABORT

$$\frac{\texttt{out\_of\_ext\_stat t} = \texttt{Some } o \qquad \textbf{abort}\, o \qquad \neg\texttt{abort\_intercepted\_stat t}}{S, C, \texttt{t} \Downarrow_s o}$$

Figure 2.5: Propagation of aborting states in JSCert

conversion, which is forbidden in pretty-big-step style. If `y` is not a normal result, then Rule RED-STAT-WHILE-2-FALSE does not apply—`vret` is a shortcut for an output terminating on a *Normal*-typed completion triple—; instead, the abortion is propagated using the rules of Figure 2.5 (there are similar rules for programs and internal reductions). The function `out_of_ext_stat` extracts an eventual output from an intermediary term—in our case, `stat_while_2` carries the result of the type conversion—; if this output is not normal, it is propagated. There are however some rules catching aborting states and we need a way to locally disable the rules of Figure 2.5: the predicate `abort_intercepted_stat` recognises these constructs. For instance, in the case of the **while** construct, *Break* and *Continue*-typed completion triples are handled by `stat_while_4` and `stat_while_5`, which are thus recognised by `abort_intercepted_stat`.

Rule RED-STAT-WHILE-2-FALSE is an axiom rule, since the corresponding ECMAScript step requires to "return (*Normal*, V, *empty*)". Variable V of ECMAScript corresponds to Variable `rv` of JSCert; it is not a completion triple, and is converted into the expected one using the type coercion below. A type coercion from a type A to a type B is an implicit function called wherever the type B was expected but the type A is given. Used properly, type coercions can sensibly increase the readability of proofs.

```
Coercion res_normal rv := {|
  res_type := restype_normal ;
  res_value := rv ;
  res_label := label_empty |}.
```

Step 2c (corresponding to Rule RED-STAT-WHILE-2-TRUE) follows the pattern of pretty-big-step: evaluate a statement (in this case `t`, or `s` in Program 2.5), and store its result. Each new pseudo-code variable becomes a parameter of a new intermediary term—in this case the parameter o_1 of `stat_while_3`. As for `stat_while_2`, the output o_1 can abort and can be propagated by Rule RED-STAT-ABORT. If no abortion happened, the computation proceeds to Step 2d, which is another conditional assignment; it is translated into a condition in Rule RED-STAT-WHILE-3. It would have been possible to split Rule RED-STAT-WHILE-3 into two rules, one in the case where `res_value R = resvalue_empty`, and one for the other case: pretty-big-step enables to break at various steps, and choices have thus to be made.

```
Inductive ext_stat :=
  | stat_basic : stat → ext_stat
  | stat_while_1 : label_set → expr → stat → resvalue → ext_stat
  | stat_while_2 :
      label_set → expr → stat → resvalue → specret value → ext_stat
  | stat_while_3 : label_set → expr → stat → resvalue → out → ext_stat
  | stat_while_4 : label_set → expr → stat → resvalue → res → ext_stat
  | stat_while_5 : label_set → expr → stat → resvalue → res → ext_stat
  | stat_while_6 : label_set → expr → stat → resvalue → res → ext_stat
  (* ... *).
```

Program 2.7: Definition of some intermediary terms in JSCert

```
Inductive red_javascript : prog → out → Prop :=

  | red_javascript_intro : forall S C p p' o o1,
      S = state_initial →
      p' = add_infos_prog strictness_false p →
      C = execution_ctx_initial (prog_intro_strictness p') →
      red_expr S C (spec_binding_inst codetype_global None p' nil) o1 →
      red_prog S C (javascript_1 o1 p') o →
      red_javascript p o

with red_stat : state → execution_ctx → ext_stat → out → Prop :=

  | red_stat_exception : forall S C extt o,
      out_of_ext_stat extt = Some o →
      abort o →
      ~ abort_intercepted_stat extt →
      red_stat S C extt o

  (* ... *)

  | red_while_2e_ii_false : forall S C labs e1 t2 rv R o,
      res_type R = restype_normal →
      red_stat S C (stat_while_1 labs e1 t2 rv) o →
      red_stat S C (stat_while_6 labs e1 t2 rv R) o

with red_expr : state → execution_ctx → ext_expr → out → Prop :=
  (* ... *)

with red_prog : state → execution_ctx → ext_prog → out → Prop :=
  (* ... *).
```

Program 2.8: Rules RED-STAT-ABORT and RED-STAT-WHILE-6-ABORT in JSCert

Step 2e is a conditional expression. The "false" case of Rule RED-STAT-WHILE-4-CONTINUE is simple: it loops back to Step 2, that is, to the intermediary term `stat_while_1`. An intermediary term `stat_while_5` is introduced in Rule RED-STAT-WHILE-4-NOT-CONTINUE for the other case. Step 2(e)iA breaks the loop, stopping the computation; this is translated by an axiom, Rule RED-STAT-WHILE-5-BREAK. The other steps proceed as expected.

The CoQ versions of the rules of Figure 2.4 are not different. The intermediary terms has to be defined first, as shown in Program 2.7. Then the rules are defined as constructors of a large inductive, as shown in Program 2.8; this program also shows the CoQ version of Rules RED-STAT-ABORT and RED-STAT-WHILE-6-ABORT. The definition of the predicate $\Downarrow_s$ for statement is separated from expressions and programs—as for $\Downarrow_i$, $\Downarrow_e$, and $\Downarrow_p$—; they are defined by mutually recursive inductives, as indicated by the **`with`**-construct of Lines 26 and 29. Note that in such a form, CoQ would accept any big-step definition: JSCERT has been written in pretty-big-step style, but there is no constraint given by CoQ to check it. Chapter 4 provides a way to enforce the constraints of pretty-big-step to apply in CoQ.

Program 2.8 also shows one additional inductive definition, with only one constructor: `red_javascript` with its introduction rule `red_javascript_intro` Line 3. Indeed, the JAVASCRIPT semantics describes both the transition system of JAVASCRIPT, but also the initial state. In JSCERT, the initial state in defined in File `JsInit.v`. This file is shown in Figure 2.3 as a common resource of both JSCERT and JSREF, but it morally should be considered as a part of JSREF as the standard is very loose about the initial environment (see Section 1.3). Program 2.9 shows an extract of `JsInit.v`: Lines 1 to 7 describe the properties of the global object defined in ECMASCRIPT, Lines 9 to 13 define its prototype and class—every object defined in the ECMASCRIPT specification have such associated definition. Then Lines 15 to 19 wraps all of these objects into the initial heap. Rule `red_javascript_intro` is the rule which sets up the initial state and execution context. Note that it performs computations before running the program `p'` Line 7: this computation—performed by the intermediary term `spec_binding_inst`—is also performed at function calls; it initialises the variables declared by the keyword **`var`** in the program `p'`. The predicate `red_javascript` enables to directly run a program in the initial state of JSREF.

2.6 JSRef: a Reference Interpreter for JavaScript

JSCERT is accompanied by a reference interpreter named JSREF. As shown in Figure 2.2, JSREF provides another way of checking JSCERT with respect to JAVASCRIPT, and thus consists in another source of trust. This interpreter has to be executable, as well as proven correct with respect to ECMASCRIPT 5, but it does not need to run fast. To ease the proof effort, JSREF has been written directly in CoQ. As a consequence, every function defined in JSREF has to be total and purely functional. In particular, the propagation of the state of the interpreted JAVASCRIPT program is explicit.

```
Definition object_prealloc_global_properties :=
  let P := Heap.empty in
  let P := write_native P "eval" prealloc_global_eval in
  let P := write_native P "parseInt" prealloc_global_parse_int in
  let P := write_native P "Object" prealloc_object in
  (* ... *)
  P.

Definition object_prealloc_global :=
  object_create_builtin
    object_prealloc_global_proto
    object_prealloc_global_class
    object_prealloc_global_properties.

Definition object_heap_initial :=
  let h : Heap.heap object_loc object := Heap.empty in
  let h := Heap.write h prealloc_global object_prealloc_global in
  (* ... *)
  object_heap_initial_function_objects h.
```

Program 2.9: Definition of the initial heap in `JsInit.v`

2.6.1 Structure of JSRef

An evaluation using a function from JSRef returns a *result*, which is either a completed computation, or a special token which states that the interpreter has reached an impossible state or that the computation did not terminate in the allocated time. As with JSCert, the type of internal reductions results depends on what is being evaluated. The type `resultof` is thus parametrised by the returned type `T` as follows.

```
Inductive resultof T :=
  | result_some : T → resultof T
  | result_impossible : resultof T
  | result_bottom : state → resultof T.
```

The special result `result_impossible` is returned by the interpreter if an invariant of JavaScript is violated—for instance if the internal method GetOwnProperty is called on a primitive value. It has been proven [Lal14] that, from a well-formed initial state, the interpreter will never return `result_impossible`. The proof relies on two main invariants: first, states and results should be well-formed (locations always point to defined objects, some global objects are defined, etc.); second, new states extend old states (in particular, old locations stay valid). This second invariant is necessary to prove the first invariant, as it implies that whatever shall happen, well-formed values stay well-formed.

Coq programs have to terminate; this is problematical as JavaScript programs may not. This problem is solved by adding a "fuel" argument, a standard technique in Coq. At each step, this argument is decremented; the execution stops when it hits zero, returning the

```
Record runs_type : Type := runs_type_intro {
    runs_type_expr : state → execution_ctx → expr → result ;
    runs_type_stat : state → execution_ctx → stat → result ;
    runs_type_prog : state → execution_ctx → prog → result ;
    runs_type_stat_while :
      state → execution_ctx → resvalue → label_set → expr → stat → result ;
    (* ... *) }.
```

(a) The definition of `runs_type`

```
Fixpoint runs max_step : runs_type :=
  match max_step with
  | 0 =>
    { runs_type_expr := fun state _ _ => result_bottom state ;
      runs_type_stat := fun state _ _ => result_bottom state ;
      runs_type_prog := fun state _ _ => result_bottom state ;
      runs_type_stat_while := fun S _ _ _ _ _ => result_bottom S ;
      (* ... *) }
  | S max_step' =>  (* max_step = 1 + max_step' *)
    { runs_type_expr := fun state => run_expr (runs max_step') state ;
      runs_type_stat := fun state => run_stat (runs max_step') state ;
      runs_type_prog := fun state => run_prog (runs max_step') state ;
      runs_type_stat_while := fun state => run_stat_while (runs max_step') state ;
      (* ... *) }
  end.
```

(b) The definition of `runs`

Program 2.10: Definition in JSRef of the potentially looping features of JavaScript

special result `result_bottom`. As several features of JavaScript can loop, more than one JSRef function need fuel. To ease the definition of JSRef, all these potentially looping functions have been gathered in a record, whose type `runs_type` is shown in program 2.10a. Each JSRef function has been implemented with an additional argument of this type. For instance the function `run_expr` has type `runs_type → state → execution_ctx → expr → result`. This enables us to define the `runs` record as a fixed point taking an integer (the fuel) as an argument and returning a record of functions, as shown in Program 2.10b. In this definition, every function takes an instantiation of the record `runs` (with less fuel) as its first parameter. Every recursive calls are then routed through this same record: each of these functions are intuitively mutually recursive. This record hides the fuel parameter, as well as the usage of `result_bottom`, avoiding to pollute JSRef. In practice, it is rare to observe `result_bottom` as the number of step can be chosen arbitrarily large (`max_int` in the case of JSRef): it never happened during the execution of all the tests of test262.

Internal reductions have results of type `resultof (specret T)` where `T` depends on the internal reduction (for instance *ToPropertyDescriptor*—`spec_to_descriptor` in JSCert—returns field descriptors). Statements and programs build outputs of type `out`, as in JSCert (see Section 2.5.2), and could thus return results of type `resultof out`. However the con-

```
Definition if_result_some (A B : Type)
    (W : resultof A) (K : A → resultof B) : resultof B :=
  match W with
  | result_some a => K a
  | result_impossible S => result_impossible S
  | result_bottom S => result_bottom S
  end.

Definition if_spec (A B : Type)
    (W : specres A) (K : state → A → specres B) : specres B :=
  if_result_some W (fun sp =>
    match sp with
    | specret_val S0 a => K S0 a
    | specret_out o =>
        if_abort o (fun _ => result_some (specret_out o))
    end).
```

Program 2.11: Two monadic operators of JSRef

structor `specret_out` of the type `specret` already carries an output; for factorisation purposes, it has been decided to reuse it. The other constructor `specret_val` can be prevented from being built by associating it with an empty type `T`. This results in the following result type for statements and programs, which is isomorphic to `resultof out`. This approach enforces every result type to be of the form `specres T = resultof (specret T)`, which simplifies the definition of monadic operators, which we now detail.

```
Inductive nothing : Type :=. (* uninhabited *)
Definition result := resultof (specret nothing).
```

2.6.2 Monadic-style Programming in JSRef

JSRef has been programmed in a *monadic style* [Wad92]. For example, to evaluate `while (e1) t2`, we first evaluate `e1`; if this evaluates to a *Normal* completion triple, we need the value produced by `e1` to continue the computation. However, if `e1` evaluates either to `result_bottom`, `result_impossible`, or to an aborting state, it has to propagate without executing the rest of the code processing `while (e1) t2`.

In JSRef, this pattern is given by the `if_spec` monadic operator. Program 2.12 shows how the `while` loop is implemented in JSRef. Consider Line 2: the first argument of `is_spec` is the computation of `e1` by `run_expr_get_value`. The second argument is the continuation, which takes as argument the new state `S1` of the program, as well as the value `v1` produced by `e1`. Program 2.11 shows the definition of `if_spec`; it uses `if_result_some` which first filters out the cases where the computation failed because of lack of fuel or because of an impossible state. If it finds any value, `if_spec` passes to the continuation `K`; otherwise it propagates aborting states.

```
1  Definition run_stat_while runs S C rv labs e1 t2 : result :=
2    if_spec (run_expr_get_value runs S C e1) (fun S1 v1 =>
3      if convert_value_to_boolean v1 then
4        if_ter (runs_type_stat runs S1 C t2) (fun S2 R =>
5          let rv' := ifb res_value R ≠ resvalue_empty
6                     then res_value R else rv in
7          let loop _ := runs_type_stat_while runs S2 C rv' labs e1 t2 in
8          ifb res_type R ≠ restype_continue
9              ∨ ~ res_label_in R labs
10         then (ifb res_type R = restype_break
11                ∧ res_label_in R labs
12               then res_ter S2 rv'
13             else (ifb res_type R ≠ restype_normal
14                   then res_ter S2 R else loop tt))
15         else loop tt)
16     else res_ter S1 rv).
17
18 Definition run_stat runs S C t : result :=
19   match t with
20   | stat_while ls e1 t2 =>
21     runs_type_stat_while runs S C ls e1 t2 resvalue_empty
22   (* ... *)
23   end.
```

Program 2.12: JSRef semantics of **`while`**-loops

Let us explain in more details Program 2.12. Argument `runs` has been explained in Section 2.6.1. Arguments `S` and `C` respectively represent the state and the execution context as explained in Section 2.5.1.2. Arguments `e1` and `t2` are the respective condition and body of the **`while`** loop. Argument `labs` is a set of labels annotating the loop (to deal with **`break`** and **`continue`** statements). Finally, `rv` is the last computed value, which will be returned in the final completion triple when the loop stops; it corresponds to the V defined Line 1 of Program 2.5. Intuitively, calling this function amounts to execute the JavaScript statement `labs: while (e1) t2` starting from heap `S` and execution context `C`. As this function is reused for the next step of the loop, the last computed value `rv` is also needed. `rv` is initially set to the empty value when called from `run_stat`, Line 21.

The body of the function works as follows. First, the condition `e1` is evaluated, and its result is captured by the continuation of Line 2. Note that this continuation only runs if the result is successful and not an abrupt termination. Following Step 2b of Program 2.5, the value `v1` is then converted to a boolean. If it is `false`, the **`else`** branch of Line 16 is taken, and the current state is returned with the last computed value `rv` (coerced to the completion triple `(normal, rv, empty)`). Otherwise, the statement `t2` is evaluated Line 4 using the monadic operator `if_ter`. This operator is similar to `if_spec`, except that it applies the continuation even if the result is an abrupt termination. This enables to check for a *Break* or *Continue* result. Lines 5 and 6 update `rv` if the result value of the statement was not empty. To proceed, the type of the completion triple is inspected. Line 15 is taken if it is of type *Continue* with its label in `labs`. Otherwise, if the result is of type *Break* with its label in `labs`, then the computation terminates as a normal result (Line 12). If the type of the result is not *Normal* (such as a *Return* or a *Break* with a label not in `labs`) then it is returned as such, otherwise the next iteration of the while loop is run (Line 14).

The description of the **while** loop in JSRef is more concise than JSCert's (shown in Figure 2.4). This observation applies to most of the constructs. Overall, the definition of JSCert is around 4,000 lines of Coq, whereas JSRef is approximately 3,000 lines.

2.6.3 Running the interpreter

As for JSCert, `JsInterpreter.v` concludes with the definition of `run_javascript`, taking a program `p` as argument and executing it in the initial state. Note the similarity between this following definition and Rule `red_javascript_intro` of Program 2.8. The function `run_javascript` is a computable function of a JavaScript interpreter in Coq; but to be easily interfaced with test suites, OCaml is more suitable than Coq.

```
Definition run_javascript runs p : result :=
  let S := state_initial in
  let p' := add_infos_prog strictness_false p in
  let C := execution_ctx_initial (prog_intro_strictness p') in
  if_void (execution_ctx_binding_inst runs S C
      codetype_global None p' nil) (fun S' =>
    runs_type_prog runs S' C p').
```

Coq provides an extraction mechanism to OCaml: it is thus possible to extract JSRef and run it against existing test suites. This mechanism is relatively simple, and can lead to very slow programs: for instance—by default—the extraction mechanism does not use OCaml's integers, but a construction of Peano arithmetic (as in Coq). Fortunately, Coq provides the ability to override the default extraction of some values and types. Of course, this feature should be used sparingly, as it comes at the expense of some trust.

JavaScript uses IEEE 754 floating-point numbers. JSCert uses the Flocq library [Mel12] to model precisely these numbers and their operations. Since the OCaml type **float** exactly corresponds to these numbers, it is safe to extract JavaScript numbers directly to OCaml **float**. Operations on numbers, such as conversion to and from Int32 types, are also provided by direct OCaml implementations.

Additionally, JSRef relies on an external parser: the development assumes the existence of a parser returning an AST or a parse error. This is expressed in Coq by the following axiom: `Axiom run_parse : string → option prog`. In order to run tests and execute the `eval` operator, `run_parse` is extracted into an OCaml function which calls an existing JavaScript parser [Hid12], then translates the output to the OCaml representation of JSCert's AST. JSCert does not use `run_parse`, but a predicate given by the following axiom: `Axiom parse : string → bool → prog → Prop`. The function `run_parse` is supposed to be coherent with the `parse` predicate.

2

2.7 Establishing Trust

The goal of JSCert is to serve as a basis for further work on JavaScript in Coq such as those presented later in this thesis, but also certified interpreters, analysers, or secure subsets. This section aims at detailing the claims of Section 2.3. JSCert's trust methodology, summarised in Figure 2.6, involves four components: the prose specification ECMAScript 5, the ECMAScript test suite Test 262, the mechanised specification JSCert, and the certified interpreter JSRef. The JSCert team established connections between ECMAScript 5, JSCert, JSRef, and Test 262 to justify that JSCert and JSRef have been designed in such a way that they can be evaluated and trusted.

JSCert has been defined as close as possible to ECMAScript 5. JSRef have been proven correct with respect to JSCert. Independently, Test 262 has been developed to cover as many aspects of ECMAScript 5 as possible, and JSRef behaves as expected on all the appropriate tests—given its coverage of ECMAScript. JSCert and JSRef can therefore be challenged through two distinct paths: through the similarity of JSCert with ECMAScript 5; and through the execution of tests by JSRef. Having these two *independent* paths significantly decreases the likelihood of bugs remaining in JSCert.

2.7.1 Trusted Base

Before explaining why one can trust JSCert, it is important to recall the implicit trusted base of JSCert. By design, the correctness of the JSCert project relies on the formal tools, libraries, parsers, as well as the translation mechanisms involved in the tool chain. JSCert is written in the Coq proof assistant using the libraries TLC [Cha10] and Flocq [Mel12]; the Coq extraction mechanism, the OCaml compiler, as well as Esprima are used to run JSRef. Also recall the code mentioned in Section 2.6.3 to bind integers and floating-point to their implementation in OCaml.

The TLC library uses some additional axioms which are not natively included in Coq. In the previous paragraph, "trusting Coq" should be understood twofold as trusting the logic behind Coq—the calculus of construction [CH88]—, and as trusting the implementation of this logic, which is Coq. Adding axioms tampers with Coq's logic. These additional axioms can unexpectedly interact with the JSCert formalisation in two aspects: in proofs built on top of JSCert and in the extraction mechanism. We now detail these two aspects.

In the case of TLC, the added axioms are the usual axioms of classical higher order logic:

- the axiom of functional extensibility: for all function f, we have $f = \lambda x.\ f\ x$. Adding functional extensibility means that η-conversion now applies on Coq's term; note that it also applies on dependently typed functions, as well as on predicates.
- the axiom of propositional extensibility; it states that equivalent propositions can be considered equals. The meaning of equality may challenge intuition in the presence of such axioms; it should be understood that if two terms are equals, then they can safely be replaced one with each other whatever the context.
- the axiom of indefinite description, which enables us to extract an element x of a proof of the form $\exists x.\ P\ x$, even if this proof is not constructive.

In particular, the propositional extensibility implies proof irrelevance:

```
Lemma proof_irrelevance : forall (P : Prop) (p q : P), p = q.
```

This property states that two proofs of the same theorem can be considered equal. This can be harmful to JSCert as the inductive of JSCert is of type `prog → out → Prop` and proof irrelevance can apply. When inverting a derivation of `red_javascript`, the path taken by the derivation matters, but this property states that it does not! The JSCert does not contain proofs using proof irrelevance before inverting a derivation, but the reader should be aware of the presence of this axiom in the Coq development.

The axiom of indefinite description conveniently provides a mechanism to perform case analysis without having to prove the decidability of each case. However, special care is required to prevent the use of the additional axioms in the computational part of the development. Coq will not warn the user when defining a term using the excluded middle—it will produce a warning during the extraction, though. Here follows an example of extracted program using TLC's classical logic. In TLC, `isTrue` is a function converting a proposition in `Prop` into a boolean—which is convenient for proofs. In this example, it has been used to produce a term which has then been extracted. During the extraction, the argument of type `Prop` has been removed, leading to an unexecutable OCaml program. Section 3.4.1 provides more details on how to solve this issue.

```
(** val isTrue : bool **)
let isTrue = failwith "AXIOM TO BE REALIZED"
```

2.7.2 Closeness to ECMAScript

As discussed in Section 2.5, JSCert has been designed to be as close as possible to ECMAScript 5: JSCert's data structures are the same as ECMAScript's, and every line of pseudo-code in ECMAScript corresponds to one or two rules in JSCert. Anyone with basic training in reading Coq specifications should be able to check the similarity between the prose of ECMAScript and the JSCert definitions. Section 2.5.2.1 shows how the specification of the `while` construct in Program 2.5 has been translated into JSCert. The other constructs are translated in the same way. This closeness does not mean that ECMAScript and JSCert look alike, but that they are close enough to relate each step to a corresponding rule, and conversely. As a consequence of its monadic style, the interpreter JSRef closely follows JSCert and thus ECMAScript (see next section). The ECMAScript community is more familiar with interpreters than formal specifications: the similarity between JSCert and ECMAScript can thus also be checked through JSRef.

JSCert intentionally differs from the prose specification at a few places. For instance, JSCert makes explicit several constructs left implicit in ECMAScript, in particular every imperative features. The state, aborting states, as well as the evaluation context and strictness flag are explicitly mentioned—whereas ECMAScript only mentions them where they are modified. Moreover, JSCert does not use "repeat" statements (as in Step 2 of Program 2.5) but rely instead on an explicit control-flow jump. It would be possible to use intermediary terms to exactly capture the "repeat" construct in JSCert's rules, but this obfuscated the inductive definition in practise. The definition of JSCert is close enough to ECMAScript 5 to be compared step by rule—this is called the "eyeball" closeness in the original paper [Bod+14]—, and thus trusted by relating it to the prose specification.

2.7.3 Correctness

The Coq development contains a proof of the correctness of JSRef with respect to JSCert. More precisely, if the JSRef interpreter evaluates a program p to an output o, then the program p is related to the output o by JSCert. The evaluation of a program in JSRef by `run_javascript` is parametrised by `runs fuel`, (shown in Program 2.10b), that is, by the maximum number `fuel` of execution steps. In cases where JSRef is stuck on a given program, the theorem does not apply. The Coq theorem is shown below.

```
Theorem run_javascript_correct : forall (n : nat) (p : prog) (o : out),
  run_javascript (runs n) p = result_some (specret_out o) →
  red_javascript p o.
```

The proof of this theorem follows the structure of the interpreter, which mainly follows the structure of JSCert; it consists of approximately 4,000 lines of Coq. Each construct is parametrised by a `runs` parameter: its correctness has thus to be passed along the proof. Program 2.13 shows the definition of this invariant. Note that none of these lemmata nor the main theorem above require the current state to be well-formed. Indeed, the interpreter always checks at run-time that the needed hypotheses are verified during.

The proof follows the structure of JSRef, construct by construct. Let us take the example of the **while** construct; Program 2.14 shows its correctness lemma. It takes as hypothesis Line 2 the correctness of its `runs` argument—in other words, the correctness of recursive calls. It then relates the function `run_stat_while` of JSRef with the intermediary term

```
Record runs_type_correct runs :=
  make_runs_type_correct {
    runs_type_correct_expr : forall S C e o,
       runs_type_expr runs S C e = o →
       red_expr S C (expr_basic e) o ;
    runs_type_correct_stat : forall S C t o,
       runs_type_stat runs S C t = o →
       red_stat S C (stat_basic t) o ;
    runs_type_correct_prog : forall S C p o,
       runs_type_prog runs S C p = o →
       red_prog S C (prog_basic p) o ;
    runs_type_correct_stat_while : forall S C rv ls e t o,
       runs_type_stat_while runs S C rv ls e t = o →
       red_stat S C (stat_while_1 ls e t rv) o ;
    (* ... *) }.
```

Program 2.13: Lemmata for each component of the runs parameter (see Program 2.10)

stat_while_1 of JSCert Line 4. It is not directly related to stat_while as Rule red-stat-while (or Step 1 of Program 2.5) performs very few computation (setting rv to resvalue_empty) and could thus been inlined. Furthermore, as explained in Section 2.5.2.1, the looping aspect of **while**-constructs is based on stat_while_1, not on stat_while. Once each construct of runs has been related to JSCert, the correctness of runs is proven by induction over the maximum number of steps fuel:

```
Theorem runs_correct : forall fuel,
  runs_type_correct (runs fuel).
```

The proof of each construct makes use of JSRef's monadic style, as well as its relation with the pretty-big-step style of JSCert shown in Section 2.6.2. Several Coq tactics have been developed to this end, in particular the run tactic; it appears for instance Line 10 of Program 2.14. This tactic is defined in Program 2.15c; it essentially consists in three steps: first, find the current JSRef monad and invert it; second, apply the given reduction rule; third, clean up the resulting context. This tactic automates the reasoning on abrupt termination cases and monad unfoldings. Thanks to the run tactic, the proof script basically consists of case analyses and the application of the right evaluation rules of JSCert.

Let us see how the run tactic works in an example. The run tactic is given Rule red-stat-while-2-true as an argument Line 10 of Program 2.14. This part of the proof corresponds to Line 4 of Program 2.12: we are in the true branch of the *if*-condition of Line 3; and Rule red-stat-while-2-true indeed applies. This rule requires to execute the term t (named t2 in Program 2.12), then proceed to the intermediary term stat_while_3. In JSRef, results passed to an intermediary term are translated to monads—in this case, the if_ter monad of Line 4 of Program 2.12. To step through the proof, we need a lemma about the behaviour of if_ter, shown in Program 2.16: there are two possibilities, either

```
Lemma run_stat_while_correct : forall runs S C rv ls e t o,
  runs_type_correct runs →
  run_stat_while runs S C rv ls e t = o →
  red_stat S C (stat_while_1 ls e t rv) o.
Proof.
  intros runs IH ls e t S C rv o R. unfolds in R.
  run_pre. lets (y1&R2&K): if_spec_post_to_bool (rm R1) (rm R).
   applys~ red_stat_while_1 (rm R2). run_post_if_spec_ter_post_bool K.
    case_if.
    run red_stat_while_2_true.
     let_name. let_simpl. applys red_stat_while_3 rv'. case_if; case_if*.
     case_if in K.
       applys red_stat_while_4_not_continue. rew_logic*. case_if in K.
         run_inv. applys* red_stat_while_5_break.
         applys* red_stat_while_5_not_break. case_if in K; run_inv.
           applys* red_stat_while_6_abort.
           applys* red_stat_while_6_normal. run_hyp*.
       rew_logic in *. applys* red_stat_while_4_continue. run_hyp*.
   run_inv. applys red_stat_while_2_false.
Qed.
```

Program 2.14: Proof of correctness for the `while` construct

```
Ltac run_select_ifres H :=
  match type of H with ?T = _ => match T with
  | @if_ter nothing _ _ => constr:(if_ter_out)
  (* ... *)
  end end.
```

(a) Getting the right behaviour lemma

```
Ltac run_pre_ifres H o1 R1 K :=
   let L := run_select_ifres H in
   lets (o1&R1&K): L (rm H).
```

(b) Extracting the needed information

```
Ltac run Red :=
  let o1 := fresh "o1" in let R1 := fresh "R1" in
  run_pre as o1 R1;
  match Red with ltac_wild => idtac | _ =>
    let o := run_get_current_out tt in
    run_apply Red o1 R1;
    try (run_check_current_out o; run_post; run_inv; try assumption)
  end.
```

(c) Applying the given reduction rule

Program 2.15: How the run tactic is defined

```
Definition isout W (Pred : out → Prop) :=
  exists o1, W = res_out o1 ∧ Pred o1.
```

(a) Stating a general property about `out`

```
1  Definition if_ter_post (K : _ → _ → result) o o1 :=
2      (o1 = out_div ∧ o = o1)
3    ∨ (exists S R, o1 = out_ter S R ∧ K S R = o).
4
5  Lemma if_ter_out : forall W K o,
6    if_ter W K = res_out o →
7    isout W (if_ter_post K o).
8  Proof.
9    introv H. destruct W as [[|o1]| | | ]; simpls; tryfalse_nothing.
10   exists o1. splits~. unfolds. destruct o1 as [|S R].
11    inverts* H.
12    jauto.
13 Qed.
```

(b) The lemma specifying the behaviour of `if_ter`

Program 2.16: Lemmata specifying monad behaviours

the argument of `if_ter` succeeds in building a result—there is then a result (Line 3 of Program 2.16b)—; or it fails, and the failure is propagated (Line 2 of Program 2.16b). Each monadic operator of JSRef has been associated a similar lemma specifying its behaviour; the tactic function `run_select_ifres` of Program 2.15a selects the needed lemma from the context. This behaviour lemma is then applied by `run_pre` (based on `run_pre_ifres` shown Program 2.15b), followed by a case analysis. Finally, the left cases are checked for aborting cases thanks to the tactic `run_post` (not shown). As a result, the monad `if_ter` has been removed from the context, and we are left with two goals: we have to prove that the argument of `if_ter` is correct[5], as well as the continuation. The case in which the argument of `if_ter` aborts has been handled by the `run` tactic.

By proving the correctness of JSRef with respect to JSCert, many typos, as well as some serious misinterpretations of ECMAScript 5, were detected and corrected. JSCert and JSRef were intentionally developed by different researchers. Despice close interaction between people, each researcher differently interpreted ECMAScript; these differences were caught during the proof and lead to discussion about how to interpret the standard. Such discussions can only increase the trust given to JSCert. The correctness proof is also a crucial part of JSCert as it enables the validation of JSCert through tests.

[5] In this particular case, it is trivial as the interpreter calls `runs_type_stat`, which is correct by the induction hypothesis. This additional goal is here automatically dealt by the tactic and does not appear.

```
1 function $ERROR (str){
2     try {
3         __$ERROR__ = __$ERROR__ + " | " + str
4     } catch (_){
5         __$ERROR__ = str
6     }
7 }
```

Program 2.17: Snippet of the JavaScript prelude of the testing architecture

```
while (1 === 1){
  __before = "reached" ;
  break ;
  __after = "dead code"
}

if (__before !== "reached")
  $ERROR ("#1: __before === 'reached'. Actual: __before === " + __before) ;
if (typeof __after !== "undefined")
  $ERROR ("#2: typeof __after === 'undefined'. Actual: typeof __after === "
    + typeof __after)
```

Program 2.18: Example of test in test262

2.7.4 Testing

JSRef has been run against the ECMAScript conformance test suite, test262 [ECM10]. This test suite requires additional functions to be defined in the initial heap. To this end, a JavaScript prelude initialising the heap with these additional functions is run prior to each test. Program 2.17 shows part of this prelude: it declares a function `$ERROR` concatenating its string argument into the global variable `__$ERROR__`; if this variable is not declared, then Line 3 throws an error, which is caught Line 4, and Line 5 properly declares this variable. It can then be checked in OCaml whether this global variable is defined after an execution: this is how we detect test failures. Program 2.18 shows a test of test262.

As said in Section 2.4.2, not all of JavaScript is covered in the JSCert project yet. In test262, there are $11,746$ tests, organised by chapters. There is no test for Chapters 1 to 5; Chapters 6 and 7 relate to the parser rather than the language; and Chapter 15 corresponds to native libraries: all these tests are not expected to pass on JSRef. There are $2,782$ tests associated with Chapters 8 to 14; out of these, JSRef passes $2,440$. These numbers changed a lot since the original JSCert paper [Bod+14] to the recent updates presented in Section 2.8; the numbers presented here are from the latter. The remaining tests mainly fail because they use **for-in** or unimplemented features of Chapter 15. More details are given in the work about the extension of JSCert [Gar+15]. Overall, JSRef successfully executed all the tests which it was expected to pass, given its coverage of ECMAScript. Section 2.8 discusses how to increase this coverage.

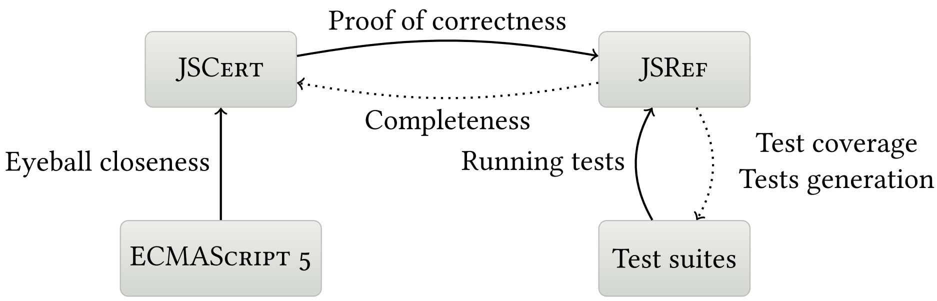


Figure 2.6: How the relation between JSCert and JavaScript is checked

Running JSRef over thousands of tests has been very useful, as it enabled to detect and fix several bugs in JSRef and JSCert. Most of these bugs were simple typos, but a few of them were more serious—such as converting a field attribute record to a data field attribute instead of a data accessor attribute (see Section 2.5.1.2). Testing helped finding bugs which the proof of correctness could not catch, such as bugs in the common files of JSCert and JSRef (see Figure 2.3). For instance, the function generating new location identifiers was found to be constant due to a small typo. As a result, creating new objects erased previously created ones, which was obviously wrong. These mistakes were quickly detected and corrected thanks to the testing architecture.

During the work on understanding ECMAScript, bugs have been found in major interpreters. For instance, all major interpreters give (different) incorrect completion values for `try-catch-finally` [Var12a; Var12c; Var12b]. The case of V8 was unexpected as dead code placed after a `try-catch-finally` construct may incorrectly change the returned value [Var12b]. This may be caused by an incorrect optimisation triggered when the program respects specific subsets of JavaScript. By altering dead code it can fall in these subsets, thus triggering the incorrect optimisation. Such bugs can be expected, as the coverage of test262 is not yet complete [Fug11]—interestingly, JSRef can help there.

2.7.5 Towards More Trust

There are two main ways to increase the trust which one can have in JSCert: by establishing the completeness of JSRef with respect to JSCert, and by evaluating the coverage of the existing test suites to complete them. These two ways correspond to the dotted arrows in Figure 2.6—arrows are drawn from trust sources: for instance, if one trusts JSCert, then the proof of correctness of JSRef can help this person to also trust JSRef.

JSRef is not proven complete (only correct) with respect to JSCert: all the behaviours of JSRef can be found in JSCert, but the converse is not proven. This means that there might be rules in JSCert enabling unnoticed behaviours, not enabled by ECMAScript. Of course, JSRef can always return `result_bottom` if it is not given enough fuel: the completeness would state that, from a starting state (possibly required to respect some invariants, such as the ones of Lallemand's proof [Lal14]), if JSCert produces a result,

```
var object_has_prop = function (l, x){
  var%some b = run_object_method (object_has_prop_, l) ;
  switch (b){
    case Coq_builtin_has_prop_default:
      var%run d = run_object_get_prop (l, x) ;
      return !full_descriptor_compare (d, Full_descriptor_undef) ;
  }
} ;
```

(a) In pseudo-JavaScript

```
let rec object_has_prop s c l x =
  let%some b = run_object_method object_has_prop_ s l in
  match b with
    Coq_builtin_has_prop_default ->
      let%run (s1, d) = run_object_get_prop s c l x in
      res_ter s1 (res_val (Coq_value_prim (Coq_prim_bool
        (not (full_descriptor_compare d Coq_full_descriptor_undef)))))
```

(b) In OCaml

```
Definition object_has_prop runs S C l x : result :=
  if_some (run_object_method object_has_prop_ S l) (fun B =>
    match B with
    | builtin_has_prop_default =>
      if_spec (runs_type_object_get_prop runs S C l x) (fun S1 D =>
        res_ter S1 (decide (D <> full_descriptor_undef)))
    end).
```

(c) In Coq

Program 2.19: A function written in the different programming languages of JSExplain

then so would JSRef, given enough fuel. This would ensure that the trust link between JSCert and JSRef is bidirectional. Such a theorem would however only be provable if JavaScript were deterministic: JSRef is a Coq function and can only return one result. Yet, it turns out that, with the exception of the loosely specified `for-in` construct, as well as some implementation dependent constructs, the standard only describes deterministic behaviours. The proof of JSCert's determinism would be an interesting way to increase the trust link between JSCert and JSRef. Alternatively, Section 4.6.3 provides directions on how to build a complete-by-construction interpreter of JSCert.

The ECMAScript community acknowledges that JavaScript's test suites are not complete with respect to ECMAScript. JSRef provides an interesting solution to this problem as it is related (through JSCert) to the ECMAScript specification. Using tool coverage programs, such as Bisect [Cle12] for OCaml, the coverage of test suites has been precisely evaluated directly on the standard [The13]. Such an analysis can also serve as a basis to generate tests, by focussing uncovered lines of JSRef. The results of these new tests could then be compared to other JavaScript interpreters. Note that the lines of JSRef producing `result_impossible` are meant to be uncovered. This alternative approach to create tests would increase the trust in JSRef by showing that it behaves like existing JavaScript implementations on tests covering its whole source code.

2.8 Extending JSCert

The methodology of JSCert could be extended to the whole JavaScript (putting the **`for-in`** construct aside, as explained in the next section): adding the rules of missing paragraphs of ECMAScript to JSCert, augmenting JSRef to implement these features, and update the proof of correctness. This would however be a significant amount of work—at least as significant as JSCert itself. Furthermore, this effort would only be worth if it could compete with the current pace of JavaScript specification: since ECMAScript 6, new versions of the standard are planned to be released every year. In addition to convince the ECMAScript community to switch to a JSCert-like specification, new ways to implement the missing features of JavaScript are being needed.

JSExplain [CSW16] is a promising path. The main goal of the JSExplain project is to provide a way for people out of the ECMAScript community to understand why a given JavaScript program acts the way it does. To do so, the project is based on a small imperative language. This imperative language shows some basic functional features, as well as all the monad-constructs of JSRef (see Section 2.6.2); these constructs would not be presented as monads, but as side-effect features. It is then possible to extract various representations of JavaScript (such as JSCert and JSRef) from a small imperative language. For instance, program 2.19a shows a snippet of this small imperative language, program 2.19b shows how it can be compiled into OCaml augmented with syntax extensions (ppx), and program 2.19c shows the equivalent definition in Coq. The Coq monads `if_some` and `if_spec` have been replaced by special constructs `var%some` and `var%run` which treat the program states (`S` and `S1` in Coq) implicitly. These three representations all carry the same amount of information, but in different ways, enabling different kinds of person to read and understand them.

When fully ready, JSExplain could export an equivalent of JSCert, JSRef, as well as a proof of their correctness. It could also be possible to extract it in other formats, such as the one described in Section 4.3. Hypothetically, it could also extract an ECMAScript-style prose specification from JSExplain: this could be a key to make the ECMAScript community adopt a JSCert-style formalisation as an official specification of JavaScript.

Waiting for JSExplain to be ready, there are others ways in which JSCert can be completed to full JavaScript. We have seen in Section 1.2.6 that most of the features of JavaScript can be implemented in a small subset of JavaScript. Mozilla Firefox and V8 have taken this into profit by implementing some of JavaScript's standard library in JavaScript [Sch12]: the parts not covered by their engine are executed by these libraries. A "`is: for (lhse in e) s`" is evaluated as follows.

1. Let $exprRef$ be the result of evaluating `e`.
2. Let $exprValue$ be $GetValue\,(exprRef)$.
3. If $exprValue$ is **`null`** or **`undefined`**, return $(normal, empty, empty)$.
4. Let obj be $ToObject\,(exprValue)$.
5. Let $V = empty$.
6. Repeat
   a) Let P be the name of the next property of obj whose *Enumerable* attribute is **`true`**. If there is no such property, return $(normal, V, empty)$.
   b) Let $lhsRef$ be the result of evaluating the `lhse` (it may be evaluated repeatedly).
   c) Call $PutValue\,(lhsRef, P)$.
   d) Let $stmt$ be the result of evaluating `s`.
   e) If $stmt.value$ is not empty, let $V = stmt.value$.
   f) If $stmt.type$ is break and $stmt.target$ is in the current label set, return $(normal, V, empty)$.
   g) If $stmt.type$ is not continue or $stmt.target$ is not in the current label set, then
      i. If $stmt$ is an abrupt completion, return $stmt$.

The mechanics and order of enumerating the properties (Step 6a) is not specified. Properties of the object being enumerated may be deleted during enumeration, [they will then] not be visited. If new properties are added to the object being enumerated during enumeration, [they] are not guaranteed to be visited in the active enumeration. A property name must not be visited more than once in any enumeration. Enumerating the properties of an object includes enumerating properties of its prototype.

Program 2.20: Specification of the **`for-in`** construct in ECMAScript 5

part of the JSCert team has followed this path [Gar+15] by adding to JSCert the features needed to run V8's JavaScript library for arrays. The new coverage of JSRef to the official test suites now includes every array tests but those using the **`for-in`** construct. The JSCert project thus keeps updating to match the real-world JavaScript.

2.9 The `for-in` Construct

The **`for-in`** construct is the major underspecified part of ECMAScript 5 [Thea]. Its specification in ECMAScript 5 is shown in Program 2.20; Steps 1 to 4 describe how the expression `e` should be executed to provide the object obj, on which the **`for-in`** construct will iterate. The interesting step is Step 6a, in which the interpreter should pick an enumerable field P of obj: it should be the *next* field—but next for which order? As indicated by the paragraph below the pseudo-code, it is not specified. But not only the order is not specified: the *iterated fields* are also underspecified.

The evaluation of `lhse` and `s` can have side effects on the iterated object; in particular, fields can be removed and added. The specification is very permissive about fields which have not yet been enumerated, but have been deleted then redefined: they can, but have not to, be enumerated. Prototype chains yield another issue: as indicated in the specification, the next field can be hidden in the prototype chain of the iterated object. But what should happen if the prototype chain is modified during the iteration? Can the fields of the old prototype chain still be iterated on? Do the fields of the new prototype chain have to be iterated on? The `for-in` construct does not specify what should happen in such cases.

The `for-in` construct—and in particular its Step 6a—comes with complex formalisation issues, as it is not specified by an operational semantics (using ECMAScript's pseudocode), but by an axiomatic semantics. Furthermore, this axiomatic semantics specifies properties about the *trace* of the execution: it explicitly requires ⌜each given field name⌝ to be iterated at most once. Specifying the `for-in` construct would thus require to trace the set of already visited field names, as well as the history of the prototype chain of the iterated object: JSCert would lose ⌜the similarity discussed in Section 2.7.2⌝. There have been some bugs [Var12c] found in major interpreters about `for-in` during the construction of JSCert. The `for-in` construct is thus a loosely defined construct with a large scale of different possible interpretations of its specification: ⌜stating that the `for-in` construct has been specified in a formal semantics⌝ is not a statement which can be taken lightly.

2.10 JSCert, JSRef, λ_{JS}, and KJS: which one to use?

In this chapter, we have described several specifications of JavaScript, with different ways to certify them. This section aims at explaining the differences between these specifications, starting by the recent KJS specification.

KJS is runnable and rule-based; it thus possesses both advantages of JSCert, with the further advantage of being simpler to manipulate. $\mathbb{K}$ is a very powerful tool to build semantics; it is furthermore accompanied by tools able to extract a Coq specification from a $\mathbb{K}$ specification—these tools are however currently experimental and do not provide the same amount of trust than writing everything directly into Coq. As JSCert, the rules of KJS closely match the ECMAScript specification, and the semantic coverage can be measured similarly to JSRef. It seems to be an excellent starting point for a formal work.

The authors of KJS claimed to have specified the `for-in` construct: given what is said in Section 2.9, closer inspection is needed. Program 2.21 is the part of KJS dealing with the `for-in` construct. Let us compare this part with the specification of `for-in` in ECMAScript 5 (Program 2.20). First, the beginning of this specification (Lines 6 to 10) closely matches the beginning of the standard (Steps 1 to 4); this closeness is comparable to JSCert's. As for JSCert, some intermediary forms have been introduced, like `@ForInAux` which represents the loop of Step 6. However, note a critical change: the list of iterated fields is computed before the loop, in Line 11 instead of during the loop (Step 6a); furthermore, ⌜the function `@EnumerateAllProperties` computing this list⌝ is deterministic, in contrary to `for-in`'s specification. $\mathbb{K}$ is not incompatible with non-determinism—the `Map`

```
syntax KResult ::= "@m" "(" Map ")"

syntax Stmt ::= "@ForIn" "(" Exp "," Exp "," Stmt ")"
rule @ForIn(L:Exp, E:Exp, S:Stmt)
  => BEGIN
    Let $e = E;
    If @OrBool(@EqVal($e, @NullVal), @EqVal($e, Undefined)) = true then {
      Return @Normal;
    } else {
      Let $o = ToObject($e);
      Let $props = @EnumerateAllProperties($o, .Map, .Set);
      Do @ForInAux(L, $o, $props, S);
    }
    END

syntax Stmt ::= "@ForInAux" "(" Exp "," K /* Oid */ "," K "," Stmt ")"
syntax Id ::= "$owner"
rule @ForInAux(_:Exp, _:Oid, @m(.Map), _:Stmt) => @Normal
rule @ForInAux(L:Exp, O:Oid, @m(P:Var |-> OP:Oid Ps:Map), S:Stmt)
  => BEGIN
    Let $desc = GetProperty(O, P);
    If $desc = Undefined then {
      Do @ForInAux(L, O, @m(Ps), S);
    } else {
      Let $owner = GetPropertyOwner(O, P);
      If $owner = OP then {
        Do %seq(%exp(%bop(%assign, L, %con(P:>String))),
             %seq(S, @ForInAux(L, O, @m(Ps), S)));
      } else {
        Do @unspecified;
      }
    }
    END

syntax KItem ::= "@EnumerateAllProperties" "(" K "," Map "," Set ")"
  rule @EnumerateAllProperties(@NullOid, TM:Map, _:Set) => @m(TM)
  rule <k> @EnumerateAllProperties(O:Oid, TM:Map, KS:Set)
    => @EnumerateAllProperties(Proto,
         #@AddProp(O, Prop, TM, KS), keys(Prop) KS) ... </k>
  <obj>
    <oid> O </oid>
    <properties> Prop:Map </properties>
    <internalProperties>
      "Prototype" |-> Proto:Oid _:Map
    </internalProperties>
  </obj>
  when O =/=K @NullOid
```

Program 2.21: The `for-in` construct in KJS construct includes an arbitrary `choice` function—, but the authors chose to build a deterministic definition, correct with respect to ECMAScript, but not complete: all expressed behaviour are valid according to ECMAScript, but some behaviours may be missing.

The goal of KJS is different from JSCert's: KJS aims at having only one object representing both the rules and an analyser, without having two definitions related by a proof of correctness. In this respect, KJS can be easily extended to support new features, including by people not familiar with proof assistants. On the other hand, JSCert aims at having a Coq-based semantics of JavaScript, which we can blindly use as the basis of any further works. For this purpose, adding the constraint of having a directly executable semantics hinders the specification of under-specified constructs (such as **`for-in`**), for which an axiomatic definition is much better suited. This is one of the reasons why JSRef has not yet been proven complete with respect to the JSCert specification, only correct.

The other specification λ_{JS} suffers from the same problem: it is correct with respect to the ECMAScript specification, but not complete. However, the variant S5 of λ_{JS} for ECMAScript 5 has now been formalised in Coq and related to the JSCert specification [Mat16]. This makes S5 a similar object than JSRef: an interpreter for JavaScript, proven correct with respect to JSCert, but not complete. They however use very different paths. JSRef is close to the JSCert specification; this makes it a potential alternative for people wanting to understand JSCert. The closeness of JSRef to the ECMAScript specification has been used to build JSExplain. λ_{JS}/S5 is composed of two parts: a compiler from JavaScript to a much simpler programming language, and an interpreter of this simpler programming language. λ_{JS} is thus a good intermediate goal to build an analyser of JavaScript, as it provides a simple programming language to analyse, to which JavaScript compiles.

The different goals of JSCert, JSRef, λ_{JS}, and KJS are thus very different, both in methodologies and in the provided guarantees. In particular, it would not be appropriate to choose one of these formalisations on the basis of the used technology alone: the authors of future works based on these formalisations should be aware of the exact provided guarantees.

2.11 Conclusion

The JSCert project has been a success in showing that modern techniques of mechanised specification can handle the complexity of JavaScript. The challenges were various: the size of the ECMAScript specification of course—JSCert contains more than 900 rules—; but also some ambiguous parts in the semantics. The paradigm change also yielded some difficulties: the contexts (the strict mode flag, evaluation context, and the state) have to be explicitly propagated, internal methods (which can return values not returnable by usual methods) did not directly fit the formalism, etc. A lot of effort has been made for JSCert to be trustable, so that certified analysers can be built on top of it. In the rest of this dissertation, we shall consider that the JSCert specification is mature enough to be used as a basis to build a certified analyser.

3 Basics of Abstract Interpretation

Myth: Computers suck because they don't do what you say.
— No! I don't download that file! It's a virus! No! Nooo!
Reality: Computers suck because they do exactly what you say.
— Ooh… `sexyladies.exe`*… This looks promising.*

Zachary Weinersmith [Wei10]

This chapter presents the framework of abstract interpretation [CC77a] as it is usually defined. It does not completely describe the framework, but provides the background required to understand the contributions of this thesis. A more detailed description of abstract interpretation can be found in Cousot's [Cou99] or Pichardie's [Pic05] works. This thesis redefines several parts of the abstract interpretation framework, in particular how derivations are built; these changes will be discussed in Chapter 4: this chapter only aims at providing some bases about abstract interpretation, which will be needed to understand the following chapters. The current chapter also presents how the important parts of the abstract interpretation framework can be implemented in Coq.

3.1 Abstract Interpretation: the Big Picture

Analysing a program consists in determining some properties about its result or its potential actions, like outputs or non-terminating behaviours. Analysers can be used to detect potential security leaks, bugs, but also to help program development. Programs may be non-deterministic, or take unknown inputs: in order to determine that something can never happen, executing the program is often not enough. Furthermore, by Rice's theorem [Ric53], most properties about programs are in general not decidable; it is thus perfectly acceptable for an analyser to abandon its analysis for some programs. There exists various methods to analyse programs, but we shall focus on abstract interpretation.

Figure 3.1 pictures how abstract interpretation works. The considered properties of the source program are semantic: they are properties about the reduction of the program in the language semantics, called the *concrete semantics*. The concrete semantics is not necessarily deterministic: there may be more than one reduction associated to a given program. The usual property aimed at by analysers is that the executed program can not reach some unsafe states—such states can be states in which invariants are broken, or any other error state. By Rice's theorem, determining whether the set of reachable states does not

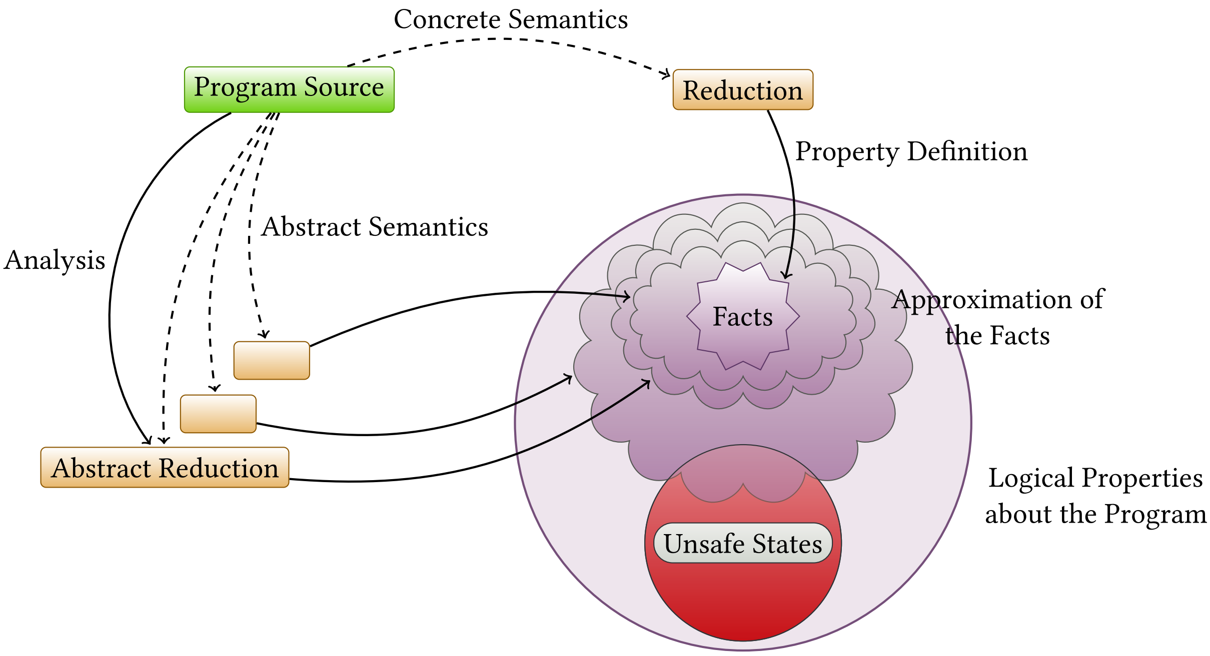
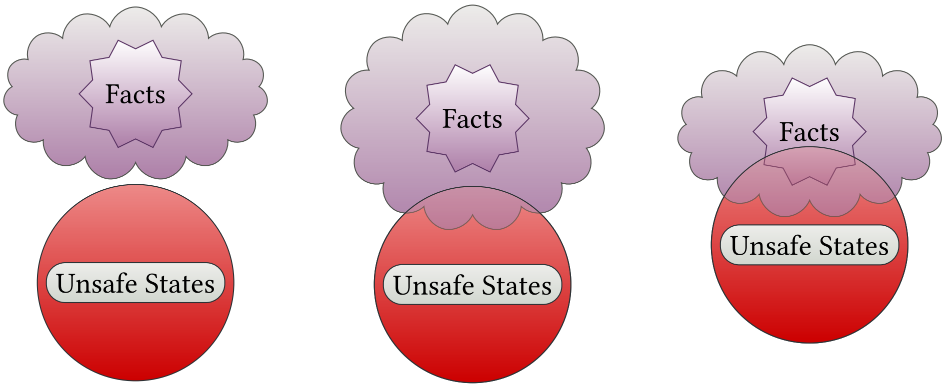


Figure 3.1: Abstract Interpretation in a Nutshell

intersect the unsafe states is an undecidable problem in general. Abstract interpretation proceeds by approximation, computing the clouds shown in Figure 3.1. These approximations must contain the reachable states of the original program: abstract interpretation aims at computing over-approximations.

Approximations may intersect the set of unsafe states, but this does not mean that the program can reach them. Figure 3.2 shows the different scenarios which can happen. First, if the approximation does not intersect the set of unsafe states, then we know—as the reachable states are included in the approximation—that the unsafe states are unreachable. Second, if the approximation does intersect the unsafe states, there are then two subcases illustrated by Figures 3.2b and 3.2c. The interesting case is the false positive, in which the approximation intersects the unsafe states, but the reachable states does not: the analysis failed to predict that the unsafe states can not happen. Abstract interpretation thus focuses in computing states which *may* happen. It is nevertheless possible to prove that something *must* happen by showing that its negation may not happen.

Let us switch back to Figure 3.1 to discuss about how these approximations are defined. Abstract interpretation proceeds by defining an *abstract semantics*; this new semantics is similar to the concrete semantics, but uses different *domains*: the precise values used in the concrete semantics have been replaced by "blurry" ones. For instance, integers can be replaced by an abstraction representing their signs: 1 is replaced by the abstract value $+$ and -1 by $-$. The abstract semantics is usually non-deterministic, leaving room for heuristics—for instance in the places where values should lose precision. The most important instance



(a) Proven safety **(b)** False positive **(c)** False negative

Figure 3.2: Different scenarios for approximations

of these heuristics consists of widening and narrowing operators [CC77a]. Sections 4.4.3 and 5.1.1 give examples of non-determinism in abstract semantics. The role of an analyser is to provide a computable version of this abstract semantics.

Abstract interpretation can be applied in various contexts. As we have seen in Section 2.1, there are various ways to specify languages. Similarly, abstract semantics can come in different forms. In this thesis, I focus on rule-based definitions: both the concrete and the abstract semantics are supposed to be made of inference rules, which are simply called *rules* in this document. Concrete and abstract rules may however be very different.

Because of the abstraction, approximations occur: there are places in which one can no longer be sure which concrete rule apply. This is a consequence of the undecidability of the analyses. Let us consider for instance that the semantics of the analysed program states that if the variable $\mathtt{x}$ is 1, then the inference rule $\mathfrak{r}_1$ applies[1], but if the variable $\mathtt{x}$ is not 1, then the inference rule $\mathfrak{r}_2$ applies. In the abstract derivation, one might have to analyse a situation in which the value of $\mathtt{x}$ is abstracted by $+$: it is not possible to know which of rule $\mathfrak{r}_1$ or rule $\mathfrak{r}_2$ applies. Applying only one of these two rules would break the over-approximation, as all the behaviours generated by the other rule would be missed. The abstract semantics has thus to differ from the concrete semantics to be sound.

3.2 Domain Structure

Abstract interpretation relies on a hypothesis: the semantics of programs can be expressed as a transition system on a given domain. This semantics is called the *concrete semantics*—as opposed to the *abstract semantics* used by analysers. Let us see how it is structured.

[1] The rule name $\mathfrak{r}$ stands for “rule”. We shall here use fraktur to denote structural characteristics of rules.

3.2.1 Concrete States

We consider a simple concrete semantics as a running example for this chapter: its syntax is shown in Figure 3.3. It features simple arithmetic expressions, an environment, and *if*-conditions. An *if*-condition tests whether an expression returns a positive value. There is no boolean in this toy language—in particular, the syntax “> 0” is part of *if*-conditions.

The semantics of the considered language is shown in Figure 3.4. We shall here only focus on a specific type of semantics, namely pretty-big-step—we already referred to it in Section 2.1.1, and next chapter formalises it in details—, but abstract interpretation is not limited to this semantics style. Note the change from Figure 2.4 about side-conditions: in order to smoothly introduce the next chapter, I adapted the rules from JSCert’s pretty-big-step to another variant of pretty-big-step. The main difference is that every premise not referring to the inductive predicate $\Downarrow$ which is being defined is written as a side-condition, as in Rules red-if-1-pos and red-if-1-neg. Also note the presence of extended terms e_e and s_e (defined in Figure 3.3), representing intermediary steps in the evaluation.

In this language, errors err are generated in Rule red-var-undef when undefined variables are accessed. The aborting rules of Figure 3.4c propagates errors in the same way than the JSCert rules of Figure 2.5: if the aborting result err is found in the semantic context, it is immediately propagated. Rule red-error-expr applies on both expressions e and extended expressions e_e, which we write e for simplicity. Similarly, Rule red-error-stat applies on both statements s and extended statements s_e. All the rules of Figures 3.4a and 3.4b request the results computed by previous rules and embedded into the current semantic context to evaluate to a non-aborting result, such as a value v or an environment E. For instance, Rule red-seq-1 applies during the execution of a sequence $s_1; s_2$, after the execution of s_1; the role of Rule red-seq-1 is to check that s_1 indeed returned an environment and not an error. The predicate **abort** looks for the aborting result err in the semantic context: the notation $C\,[err]$ represents any semantic context carrying the error err. This language is meant to be updated along this dissertation, and this will change the possible semantic contexts—for instance, Figure 4.2 will add semantic contexts for loops—: the aborting rules will be assumed to be updated accordingly.

Concrete domains are formed by the set of states transferred along the derivation trees. In this case, expressions return either values in $Val = \mathbb{Z}$ or the error err and takes environments in $Env = Var \rightharpoonup_{fin} Val$ (finite maps from variables to values) as semantic context. Statements take environments and return either an environment or the error err.

Definition 3.1 (Concrete Domains). We define the following sets.

- $Val = \mathbb{Z}$;
- $error = \{err\}$;
- $Env = Var \rightharpoonup_{fin} Val$, the finite maps from Var to Val;
- $Out_e = Val + error$, the expression outputs;

$$
\begin{aligned}
e &::= c \in \mathbb{Z} \\
&\mid \mathsf{x} \in \mathit{Var} \\
&\mid e_1 + e_2
\end{aligned}
\qquad
\begin{aligned}
s &::= \mathit{skip} \\
&\mid s_1; s_2 \\
&\mid \mathsf{x} := e \\
&\mid \mathit{if}\ (e > 0)\ s_1\ s_2
\end{aligned}
\qquad
\begin{aligned}
e_e &::= \cdot +_1 e \\
&\mid \cdot +_2 \cdot
\end{aligned}
\qquad
\begin{aligned}
s_e &::= \mathsf{x} :=_1 \cdot \\
&\mid \cdot ;_1 s_2 \\
&\mid \mathit{if}_1\ s_1\ s_2
\end{aligned}
$$

Figure 3.3: A simple language featuring variables and arithmetic expressions

$$
\frac{}{E, c \Downarrow c}\ \textsc{red-const}
\qquad
\frac{}{E, \mathsf{x} \Downarrow E[\mathsf{x}]}\ \mathsf{x} \in \mathit{dom}(E)\ \textsc{red-var}
\qquad
\frac{}{E, \mathsf{x} \Downarrow \mathit{err}}\ \mathsf{x} \notin \mathit{dom}(E)\ \textsc{red-var-undef}
\qquad
\frac{E, e_1 \Downarrow r \qquad E, r, \cdot +_1 e_2 \Downarrow r'}{E, e_1 + e_2 \Downarrow r'}\ \textsc{red-add}
$$

$$
\frac{E, e_2 \Downarrow r \qquad E, v_1, r, \cdot +_2 \cdot \Downarrow r'}{E, v_1, \cdot +_1 e_2 \Downarrow r'}\ \textsc{red-add-1}
\qquad
\frac{}{E, v_1, v_2, \cdot +_2 \cdot \Downarrow v_1 + v_2}\ \textsc{red-add-2}
$$

(a) Expressions

$$
\frac{}{E, \mathit{skip} \Downarrow E}\ \textsc{red-skip}
\qquad
\frac{E, s_1 \Downarrow r \qquad r, \cdot ;_1 s_2 \Downarrow r'}{E, s_1; s_2 \Downarrow r'}\ \textsc{red-seq}
\qquad
\frac{E, s_2 \Downarrow r}{E, \cdot ;_1 s_2 \Downarrow r}\ \textsc{red-seq-1}
\qquad
\frac{E, e \Downarrow r \qquad E, r, \mathsf{x} :=_1 \cdot \Downarrow r'}{E, \mathsf{x} := e \Downarrow r'}\ \textsc{red-asn}
$$

$$
\frac{}{E, v, \mathsf{x} :=_1 \cdot \Downarrow E[\mathsf{x} \leftarrow v]}\ \textsc{red-asn-1}
\qquad
\frac{E, e \Downarrow r \qquad E, r, \mathit{if}_1\ s_1\ s_2 \Downarrow r'}{E, \mathit{if}\ (e > 0)\ s_1\ s_2 \Downarrow r'}\ \textsc{red-if}
\qquad
\frac{E, s_1 \Downarrow r}{E, v, \mathit{if}_1\ s_1\ s_2 \Downarrow r}\ v > 0\ \textsc{red-if-1-pos}
$$

$$
\frac{E, s_2 \Downarrow r}{E, v, \mathit{if}_1\ s_1\ s_2 \Downarrow r}\ v \leqslant 0\ \textsc{red-if-1-neg}
$$

(b) Statements

$$
\frac{}{\sigma, e \Downarrow \mathit{err}}\ \mathbf{abort}\,\sigma\ \textsc{red-error-expr}
\qquad
\frac{}{\sigma, s \Downarrow \mathit{err}}\ \mathbf{abort}\,\sigma\ \textsc{red-error-stat}
\qquad
\frac{\sigma = C[\mathit{err}]}{\mathbf{abort}\,\sigma}
$$

(c) Aborting rules

Figure 3.4: A simple semantics featuring variables and arithmetic expressions

- $Out_s = Env + error$, the statement outputs.

Extended terms are also associated semantic contexts and results—for instance, the extended term $if_1\ s_1\ s_2$ takes as inputs an environment and an expression output. As for JSCert (see Figure 2.5), errors are propagated by Rules red-error-stat and red-error-stat.

In this dissertation, we use the following notations for environments—and more generally for each map. Given an environment E and a variable x, we write $E[\mathsf{x}]$ the value of x in the environment E. Given an additional value v, we write $E[\mathsf{x} \leftarrow v]$ for a new environment E' such that $E'[\mathsf{x}] = v$, $dom(E') = dom(E) \cup \{\mathsf{x}\}$, and $E'[\mathsf{y}] = E[\mathsf{y}]$ for all $\mathsf{y} \in dom(E) \setminus \{\mathsf{x}\}$. In particular, $E[\mathsf{x} \leftarrow v]$ does not change E, but produces a new environment. We write $E \setminus \mathsf{x}$ for an environment equal to E except that it has no binding for x. Given two environments E_1 and E_2 with disjoint domain, the notation $E_1 \uplus E_2$ stands for the environment E such that $dom(E) = dom(E_1) \uplus dom(E_2)$ and for all x, $E[\mathsf{x}]$ is either $E_1[\mathsf{x}]$ or $E_2[\mathsf{x}]$, depending which is defined. The empty environment is written ϵ, and an environment mapping x to v_1 and y to v_2 is written $\{\mathsf{x} \mapsto v_1, \mathsf{y} \mapsto v_2\}$.

3.2.2 Abstract Lattice

Abstract interpretation is based on abstract domains related to these concrete domains—for instance values Val can be abstracted by $Val^\sharp = Sign = \{\bot, -, 0, +, -_0, \pm, +_0, \top_\mathbb{Z}\}$, which tracks value signs. There exist more useful and precise domains (such as intervals [CC77b]) but to avoid dispersion we shall consider the sign domain when possible. The techniques presented in this dissertation work with any other abstract domain.

We can lift the domain $Val^\sharp$ abstracting basic values to abstract other concrete domains. For instance, the domain of expressions outputs Out_e can be abstracted as $Out_e^\sharp = Val^\sharp \times error^\sharp$, where $error^\sharp = \{err^\sharp, \overline{err}^\sharp\}$: the abstract output $(v^\sharp, \overline{err}^\sharp)$ represents a non-error output represented by $v^\sharp$, whilst $(v^\sharp, err^\sharp)$ represents either err or a value abstracted by $v^\sharp$ (see Section 3.5.1 for more details). Abstract values are usually annotated by the symbol $\sharp$, but $\sharp$ is not a function: $v^\sharp$ is not built from an hypothetical concrete value v. Each state σ of the concrete domain can be abstracted following this process into an abstract state $\sigma^\sharp$. There also exist more elaborate abstractions storing relations between values. Such domains are called *relational*—the octagon domain [Min06b] is a famous instance.

The advantage of manipulating abstract domains is the ability to perform approximations: some elements in the lattice represent more concrete elements than others. To this end domains are usually supposed to be equipped with a (decidable) structure of lattice:

- They are equipped with a decidable structure of partially ordered set (poset)—in other words, with a partial order $\sqsubseteq$ and its decision procedure. Intuitively, if $v_1^\sharp \sqsubseteq v_2^\sharp$, then $v_2^\sharp$ represents more concrete elements than $v_1^\sharp$, which is thus more precise.

- They are equipped with a binary least upper bound $\sqcup$, pronounced "join", and a binary greatest lower bound $\sqcap$, pronounced "meet". These operations must respect the following properties.

$$\forall x, y.\ x \sqsubseteq x \sqcup y \tag{3.1}$$

$$\forall x, y.\ y \sqsubseteq x \sqcup y \tag{3.2}$$

$$\forall x, y, z.\ x \sqsubseteq z \to y \sqsubseteq z \to\ x \sqcup y \sqsubseteq z \tag{3.3}$$

$$\forall x, y.\ x \sqcap y \sqsubseteq x \tag{3.4}$$

$$\forall x, y.\ x \sqcap y \sqsubseteq y \tag{3.5}$$

$$\forall x, y, z.\ z \sqsubseteq x \to z \sqsubseteq y \to\ z \sqsubseteq x \sqcap y \tag{3.6}$$

Given any set A, its powerset $\mathcal{P}(A)$ can be ordered as a lattice: the order relation is the set inclusion $\subseteq$, the join operator is the set union $\cup$, and the meet operator the set intersection $\cap$. Such a lattice is the most precise abstract domain which can be built: a set $S \in \mathcal{P}(A)$ represents exactly its elements. It is usually not a good abstract representation, as some sets are not finitely representable. Domains such as $Sign$ are more concise (although less precise), leading to a decidable structure; such structure can even be efficient in association with techniques such as widening and narrowing. Choosing the right domains is often a trade-off between preciseness and efficiency.

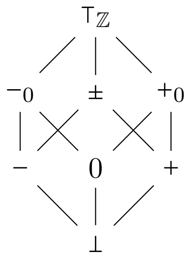
Lattices can be represented through Hasse diagrams. Figure 3.5a represents the Hasse diagrams of $Val^\sharp = Sign$. Each link between two abstract values $v_1^\sharp$ and $v_2^\sharp$—$v_2^\sharp$ being upper than $v_1^\sharp$— in such a diagram expresses that $v_1^\sharp \sqsubseteq v_2^\sharp$. The order $\sqsubseteq$ of the lattice can be inferred from such diagram by taking the transitive closer of each of these steps.

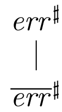
The sign domains is a complete lattice: each of its subset S has a least upper bound $\bigsqcup S$. A complete lattice also comes with a greatest lower bound $\bigsqcap$, defined as follows.

$$\bigsqcap S = \bigsqcup \{x \mid \forall y \in S, x \sqsubseteq y\}$$

Every finite lattice is also a complete lattice. Indeed, the properties 3.1, 3.2, and 3.3 imply that $\sqcup$ is commutative and associative; it is thus possible to fold $\sqcup$ on any subset from an initial value. From these foldings, it is possible to define $\bigsqcup$ given a particular element named $\bot$ and pronounced "bottom"; this particular element is the smallest of the lattice and is neutral for $\sqcup$ and thus for $\bigsqcup$: its presence or absence in the considered subset does not change the result. Such an element necessarily exists in a finite lattice.

This proof sketch introduces two interesting elements of a complete lattice: the upper bound of all elements, named $\top$ and pronounced "top", as well as the lower bound of all elements, named $\bot$ and pronounced "bottom". When the context is not clear, the top and bottom elements of a lattice L can be annotated as $\top_L$ and $\bot_L$. As $\top$ is greater than every abstract elements, it represents all the concrete elements: if an analysis states that the output of a program can be abstracted by $\top$, it provides no information about what can





(b) The error domain

(a) The *Sign* domain

Figure 3.5: Some Hasse diagrams

$$\gamma(\top_{\mathbb{Z}}) = \mathbb{Z} \qquad \gamma(\pm) = \mathbb{Z}^{\star} \qquad \gamma(+_0) = \mathbb{Z}_{+} \qquad \gamma(+) = \mathbb{Z}_{+}^{\star}$$
$$\gamma(-_0) = \mathbb{Z}_{-} \qquad \gamma(-) = \mathbb{Z}_{-}^{\star} \qquad \gamma(0) = \{0\} \qquad \gamma(\bot) = \varnothing$$

Figure 3.6: Definition of the concretisation function for the sign domain

happen in the program. On the contrary, $\bot$ usually[2] represents no concrete state: if an analysis states that the output of a program is abstracted by $\bot$, we know that the program never outputs—either because it loops or because no concrete derivation exist. This leads us to consider what are the concrete objects represented by a given abstract element.

3.2.3 Concretisation Functions

We have assumed that abstract values were representations of concrete values. In practise, we relate each abstract value $v^\sharp$ to a set of concrete values $\gamma\left(v^\sharp\right)$. More precisely, the lattice of the abstract domain is related to the lattice of the powerset of the concrete domain by a Galois connection (γ, α). This means that given any set of concrete values S and abstract value $v^\sharp$, we have the following equivalence:

$$\alpha(S) \sqsubseteq v^\sharp \iff S \subseteq \gamma\left(v^\sharp\right)$$

Intuitively, $\gamma\left(v^\sharp\right)$ is the set of concrete values v represented by $v^\sharp$. Figure 3.6 shows the definition of the concretisation function γ for the sign domain $Val^\sharp$. Conversely, given a set S of concrete values, $\alpha(S)$ is the most precise abstract value representing them. I did not define what are the exact types of γ and α. The reason is that this dissertation describes a lot of abstract domains, associated with their corresponding concretisation functions. I similarly use $\sqsubseteq$ for each lattice order, without specifying which lattice I am considering: it would make notations heavier without adding any useful clarification.

[2] It is not required by the correctness of concretisation functions—as it is not required for the soundness—, but it is very common in practise. It is also a good intuition of what $\bot$ represent. All the domains built during this thesis have a bottom element with empty concretisation.

The concretisation function γ is used to express the soundness of an analysis (see Section 4.4.3): an analysis returning the result $r^\sharp$ is sound if every concrete result returned by the analysed program is in $\gamma\left(r^\sharp\right)$. Symmetrically, the abstraction function α is used to express the preciseness of an analysis: if R is the set of possible results of a program, then $\alpha\left(R\right)$ is the most precise abstract value which a sound analyser can return on it.

3

I shall not extend what are the full consequences of the Galois connection of (γ, α), as they have not been fully used in this dissertation. Precision is indeed not a goal of this thesis—or more precisely, my results can sometimes be precise, but they are never *proven* to be precise. Focusing on soundness allows us to remove some hypotheses of the manipulated structures, and thus diminish the proof effort. Similarly to α, the hypotheses 3.3 and 3.6 of the lattice structure concerns preciseness and can be removed. Removing these hypotheses does not mean that they will be violated in the abstract domains built in this dissertation, only that they will not (have to) be proven.

3.2.4 Restriction of the Axioms of Abstract Interpretation

Here follow the properties about abstract domains with respect to their concrete domains requested in this dissertation:

- Abstract domains are equipped with a poset structure. This structure should be partially decidable: there exists a partial boolean function such that for each $v_1^\sharp$ and $v_2^\sharp$, if this function is defined and its result is true then $v_1^\sharp \sqsubseteq v_2^\sharp$.
- Abstract domains are equipped with two computable partial operators $\sqcup$ and $\sqcap$ respecting the properties 3.1, 3.2, 3.4, and 3.5 where they are defined.

$$\forall x, y.\ x \sqcup y \text{ defined} \implies x \sqsubseteq x \sqcup y \tag{3.1 revisited}$$

$$\forall x, y.\ x \sqcup y \text{ defined} \implies y \sqsubseteq x \sqcup y \tag{3.2 revisited}$$

$$\forall x, y.\ x \sqcap y \text{ defined} \implies x \sqcap y \sqsubseteq x \tag{3.4 revisited}$$

$$\forall x, y.\ x \sqcap y \text{ defined} \implies x \sqcap y \sqsubseteq y \tag{3.5 revisited}$$

- Abstract domains are equipped with a concretisation function γ from the abstract domain to the powerset of the concrete domain compatible with the poset order of the abstract domain:

$$\forall v_1^\sharp, v_2^\sharp.\ v_1^\sharp \sqsubseteq v_2^\sharp \implies \gamma\left(v_1^\sharp\right) \subseteq \gamma\left(v_2^\sharp\right) \tag{3.7}$$

Note that the two operators $\sqcup$ and $\sqcap$ are supposed to be *partial* operators: they are allowed to fail merging two results. In such failing cases, the construction of an abstract derivation also fails, preventing any unsound derivation to be constructed. Of course, such a failure would mean that an analyser aborts a program analysis, which is usually not a good action

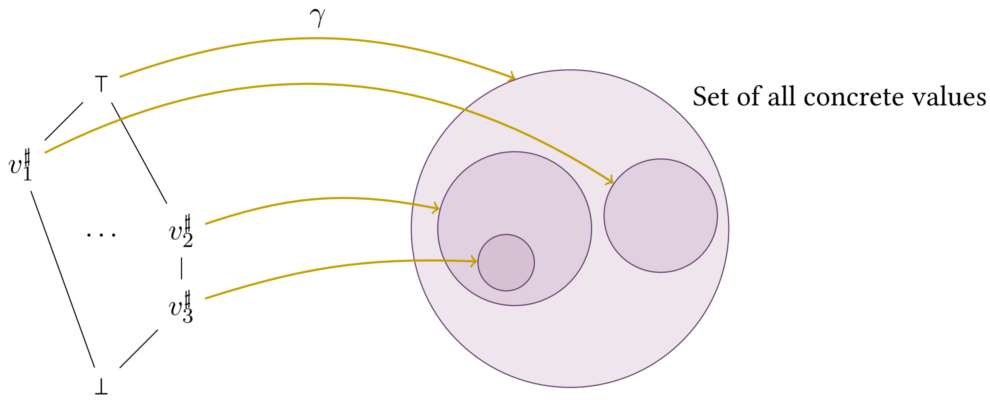


Figure 3.7: Concretisation relation between an abstract and a concrete domain

from an analyser. This issue can be compensated by constructing a symbolic completion of the original domain (see Section 3.5.2). The point is not to build aborting analyses, but to avoid having to prove that analyses never abort.

Figure 3.7 pictures the constraint on the concretisation function: the poset structure is at left and the concrete domain is at right. The poset structure is ordered in this figure as in a Hasse diagram; however the concrete domain at the right is not ordered: the different regions include concrete elements. Regions are ordered with the inclusion relation $\subseteq$. In this case, the abstract domain contains a $\top$ and a $\bot$ value, but this is not mandatory.

3.3 Abstract Interpretation of Big-step Semantics

Abstract interpretation provides a systematic way of building abstract semantics from a concrete semantics and an abstract domain. It consists of the following steps:

- choose an abstract domain;
- define an abstract semantics over this abstract domain;
- show that its abstract executions are sound with respect to the concrete executions;
- program an *analyser* building an abstract execution among the possible ones. This analyser is sound by construction. Its precision depends on the chosen execution.

Some analysers do not build abstract executions, but are related to the abstract semantics by a soundness proof. This soundness proof is technically an analyser sound by construction: from any execution of the associated analyser, it builds an abstract execution.

This is how Cousot [Cou99] and Midtgaard and Jensen [MJ08] systematically build abstract semantics from transition systems. Some [Cac+05; Jou+15] even defined an abstract semantics in Coq for a non-trivial language (the C♯minor language in this example). These abstractions usually separate side-effects-free programs (usually named *expressions*) from programs with potential side-effects (usually named *statements*). It is possible to make use

$$
\dfrac{\dfrac{\vdots}{E^\sharp, e \Downarrow +} \qquad \dfrac{\dfrac{\vdots}{E^\sharp, s_1 \Downarrow r_1^\sharp}}{E^\sharp, +, \mathit{if}_1\ s_1\ s_2 \Downarrow r_1^\sharp}}{E^\sharp, \mathit{if}\ \ (e > 0)\ s_1\ s_2 \Downarrow r_1^\sharp} \quad \sqsubseteq \quad \dfrac{\dfrac{\vdots}{E^\sharp, e \Downarrow \top_\mathbb{Z}} \qquad \dfrac{\dfrac{\vdots}{E^\sharp, s_1 \Downarrow r_1^\sharp} \qquad \dfrac{\vdots}{E^\sharp, s_2 \Downarrow r_2^\sharp}}{E^\sharp, \top_\mathbb{Z}, \mathit{if}_1\ s_1\ s_2 \Downarrow r_1^\sharp \sqcup r_2^\sharp}}{E^\sharp, \mathit{if}\ \ (e > 0)\ s_1\ s_2 \Downarrow r_1^\sharp \sqcup r_2^\sharp}
$$

Figure 3.8: An approximation of an abstract derivation tree

of the particularities of these two kinds of programs (expressions and statements). The semantics of expressions is usually straightforward, and their semantics is usually defined in a style following the derivation structure, such as big-step style (see Section 2.1.1). Conversely, because statements do not always terminates, they are usually specified in small-step style. This separation is problematic for JavaScript in which every program has potential side-effects: usual approaches would choose the small-step style for JavaScript, but JSCert is written in (pretty-)big-step.

The principles behind abstract interpretation of big-step semantics have been studied by Schmidt [Sch95]; they form the basis of the formalisation of Chapter 4. The idea behind these principles is to lift the connection between the concrete and abstract worlds (usually a Galois connection) to derivations trees. Given the restrictions of Section 3.2.4, this amounts to define a poset and a concretisation function for abstract derivation trees. Schmidt introduced a precise definition of what a semantic derivation tree is: it is a derivation tree obtained from applying the inference rules of a big-step semantics to a term. Semantic derivation trees result in *concrete judgements* of the form $\sigma, t \Downarrow r$, where σ is a semantic context, t a term, and r a result. An *abstract semantic tree* (also called *abstract derivation*) is then defined to be a semantic derivation tree where the values at the nodes are in the abstract domain. Schmidt then defined how derivations can be approximated. A derivation can be approximated either by approximating one of its judgements or by adding branches to it. Approximating a judgement of a derivation implicitly implies to propagate the effects: to be a valid derivation tree, the new tree has to follow the inference rules. Figure 3.8 shows an instance of this order: the abstract value $+$ has been approximated into $\top_\mathbb{Z}$. Because of this approximation, a new branch had to be added in the abstract derivation tree to cover the new non-positive case. Schmidt showed that the complete lattice of semantic contexts and results can be lifted to abstract derivations[3]: abstract derivations form a complete lattice when equipped with the above order $\sqsubseteq$. Relating concrete and abstract derivations in such a way provides strong principles on how abstract derivation should be defined and proven sound.

[3] Schmidt manipulated derivations augmented with a top and a bottom element—respectively written Δ and Ω—which could appear as subtrees of derivations. Such derivations enable to define greatest fixed points without using coinduction. These formalisation choices are not crucial to understand this dissertation.

From the concretisation function of abstract input states $\sigma^\sharp$ and abstract results $r^\sharp$, it is possible to define the concretisation of an abstract judgement $\sigma^\sharp, t \Downarrow r^\sharp$. The concretisation relation between abstract and concrete derivation trees is defined as follows. A concrete and an abstract derivations π and $\pi^\sharp$ are related if the conclusion statement of π is in the concretisation of the conclusion of $\pi^\sharp$, and for each sub-derivation of π, there exists a corresponding abstract sub-derivation of $\pi^\sharp$ which covers it. Intuitively, an abstract derivation covers a concrete derivation if the latter can be "recognised" in the former.

There are several ways in which the coverage of abstract derivations can be be ensured. One way is to add a number of ad-hoc rules. For example, it is common for inference-based analyses to include a rule such as Rule IF-ABS below, which covers the execution of both branches of an *if*-construct. Section 3.6 explains the problem with such rules, and Section 4.4.1 proposes an alternative.

$$\textsc{if-abs}\ \frac{E^\sharp, s_1 \Downarrow r_1^\sharp \qquad E^\sharp, s_2 \Downarrow r_2^\sharp}{E^\sharp, \mathit{if}\ (e > 0)\ s_1\ s_2 \Downarrow r_1^\sharp \sqcup r_2^\sharp}$$

Some programs can loop a non-deterministic number of step: there may be no bound on the maximum size of a concrete derivation for a given program. As an abstract derivation has to include all of the corresponding concrete derivations, abstract derivation trees may have to be infinite. An analyser can still terminate by identifying an invariant in the derivation tree. Whatever the invariant used by the analysis, it is sound if the returned derivation belongs to the set of abstract derivations trees. It is important for Chapter 4 to understand that each of these abstract derivation trees is sound. When Schmidt defined his abstract interpreter, he considered all the abstract derivation trees whose conclusion was in the form $\sigma^\sharp, t \Downarrow r^\sharp$ for a given $\sigma^\sharp$ and t, then took the smallest derivation tree (which is possible because derivation trees form a complete lattice). The smallest derivation tree is guaranteed to build the most precise result $r_0^\sharp$, but any other result $r^\sharp$ produced by another derivation tree would still be sound, as by construction we have $r_0^\sharp \sqsubseteq r^\sharp$, and thus $\gamma\left(r_0^\sharp\right) \subseteq \gamma\left(r^\sharp\right)$. A less precise result $r^\sharp$ might however be much simpler to find than the most precise result $r_0^\sharp$: we shall not limit ourselves in this dissertation to the most precise result, but will accept any sound result.

Chapter 4 describes how this way of building abstract semantics has been extended to ease the proof of soundness of the abstract semantics. This proof has been defined in the Coq proof assistant. We now present how the different mathematical notions seen in this chapter are expressed in the Coq files accompanying this dissertation [Bod16].

3.4 Practical Abstract Interpretation in Coq

The goal of Coq is to build very rigorous proofs. This can sensibly hinder the proof effort as each fact—however "trivial" or evident they may be—has to be proven. Type classes [SO08] provide a practical solution to this problem: Coq is equipped with a mechanism looking for some specific instances as need. Such instances can be defined by the user to adapt the needs of a particular development. Instances are a way to handle implicit proof in Coq. Let us see how they work on some examples.

3.4.1 Decidable Instances

The Coq development described in this dissertation is based on the TLC library [Cha10]. As we have seen in Section 2.7.1, this library is based on classical logic. In particular, TLC provides the following reflection mechanism: the function `isTrue : Prop → bool` takes a property and returns the `true` boolean if and only if the given property is true. Of course, this function is not extractable, as not all properties are decidable.

When we need to extract such a test, we have to show that a given property `P` is decidable. This amounts to define a decision procedure[4] for this property `P`. Defining such a procedure (and proving it) can be cumbersome in a lot of cases, in particular when they stack on top of each other. For instance, to look up a value into an associative list, we have to loop through the list looking for a given key: keys have to be proven comparable, or in other words, the equality of keys has to be proven decidable. These procedures are not difficult to define or prove, but when in the middle of a big definition such as JSRef, they can stop ourselves from the formalisation effort.

In order to bypass this problem, we use the following type class specifying a decidable predicate. This class is composed of two elements: Line 2 contains a boolean `decide`, and Line 3 specifies how this boolean behaves. In this case, the boolean `decide` should be equivalent to `isTrue`, but in a computable way. This approach can be considered as a small-scale reflection, packing together a predicate and a boolean function.

```
1 Class Decidable (P : Prop) := make_Decidable {
2   decide : bool ;
3   decide_spec : decide = isTrue P }.
```

Once the decidability of the property has been defined, type classes enable to only refer to the tested property, and to forget about the precise instantiation needed to prove it. The unextractable expression `If x = y then e1 else e2` can now be written `if decide (x = y) then e1 else e2`, shortened into `ifb x = y then e1 else e2`. At each occurrence of `decide`, Coq will look for known instances of the `Decidable` class. If found, Coq will transparently accept this definition: the only change from the user's perspective is to

[4] Note that this property can be parametrised by some values: the intuition is that `P` is a predicate.

replace `If` by **`ifb`**. Internally, Coq builds a boolean which can be extracted. Extraction then builds a term of the form **`if`** `comparable_instance x y` **`then`** `e1` **`else`** `e2`, where `comparable_instance` is the instance built by the type class mechanism. We have already transparently encountered the **`ifb`**-construct—for instance Line 4 of Program 2.1b.

This simplifies the definition of terms and removes the need to prove their correctness. For instance, the comparison of references (see Section 2.4.1) has been implemented in JSRef as below. This definition is readable because of type classes. Furthermore, we can directly infer from this definition that it returns `true` if and only if all these equalities hold.

```
Definition ref_compare r1 r2 : bool :=
  decide (ref_base r1 = ref_base r2 ∧
    ref_name r1 = ref_name r2 ∧
    ref_strict r1 = ref_strict r2).
```

Once extracted, we can see that Coq reused several type class instances which have been defined in TLC and in the JSCert development, such as `string_comparable`.

```
let ref_compare r1 r2 =
  and_decidable (ref_base_type_comparable r1.ref_base r2.ref_base)
    (and_decidable (string_comparable r1.ref_name r2.ref_name)
      (bool_comparable r1.ref_strict r2.ref_strict))
```

Another class frequently used in the development is the `PartiallyDecidable` class of Program 3.1a. It is very similar to the `Decidable` class, with one difference: instead of the `try_to_decide` to be equivalent to the trueness of the given property `P`, it only implies it. If `try_to_decide` is `false`, then it provides no information about the property `P`; if it is `true`, the property `P` has to be true. The `PartiallyDecidable` class is useful to build sound analysers without having to prove that they are precise. For instance, an analyser could choose to be precise if a given property is true, and fall back to a less precise way if it is not: the other way is sound in both cases, just less precise. Invoking `try_to_decide` instead of `decide` can be a way to be precise when needed, but not when the proof effort is huge. Program 3.1b is an example of type class instance which can be used by Coq to infer new instances. In this case, the instance states that a decidable property is partially decidable. The proof uses the precise `decide P` for the value of `try_to_decide`: this instance is as precise as the previous one, we only lost the proof of its precision in the process.

Instances such as the one shown in Program 3.1b extend the type classes inferred by Coq. This is especially useful when defining mathematical structures, as they are often composed of substructures. For instance, a poset provides a $\sqsubseteq$ operator, which is by hypothesis decidable: Coq will automatically use this hypothesis wherever it is needed to transparently define new type class instances.

```
Class PartiallyDecidable (P : Prop) : Type := PartiallyDecidable_make {
    try_to_decide : bool ;
    try_to_decide_spec : try_to_decide → P }.
```

(a) Class definition

```
Global Instance Decidable_PartiallyDecidable : forall P,
    Decidable P →
    PartiallyDecidable P.
  introv D. applys PartiallyDecidable_make (decide P). rew_refl~.
Defined.
```

(b) Relation to the `Decidable` class

Program 3.1: The `PartiallyDecidable` class

3.4.2 The Poset Class

Posets are often built by composing several smaller posets; this can make the definition of their operations complex and difficult to read. Cachera and Pichardie [Pic08; CP10] provide some useful instances which have been used all along this thesis's development.

The data structures used to represent elements are usually richer than the represented mathematical object: sets can for instance be represented by lists or trees, but the order on such lists is not relevant to the represented set. Quotients are thus frequent when building these structures; this makes Coq's minimal equivalence relation = of limited use. To ease the development, it is thus preferable to be parametrised by a (decidable) equivalence relation. The type class `EquivDec.t A` of Program 3.2a equips its argument type `A` by an equivalence relation noted $=\sharp$. This relation is supposed to be decidable.

The class `PosetDec.t A` of Program 3.2b defines the structure of decidable poset over the type `A`; it provides an instance of `EquivDec.t A` (and thus a $=\sharp$ operator), as well as a (decidable) order relation $\sqsubseteq\sharp$. When writing a property involving $\sqsubseteq\sharp$, Coq shall look for corresponding instances. This greatly simplifies notations and reasoning.

Program 3.3 shows how the fact that a domain has a $\top$ and a $\bot$ elements can be expressed as a type class. These two structures take as a parameter their corresponding poset, in contrary to `PosetDec.t` which provides its equivalence relation. This parametrisation enables the usage of these classes separately: not all structures have both a $\top$ and a $\bot$ elements; and these elements are only needed in specific situations. These choices are more driven by usability rather than by an intrinsic mathematical property.

For each of these structures, we have assumed that the operations $=\sharp$ and $\sqsubseteq\sharp$ are decidable. For some structures—in particular the ones described in Chapter 6—, this can hinder the definition of an abstract domain. We shall thus sometimes use the alternative definition of posets below. It features the same properties, except its decidability, and is noted $\sqsubseteq$. Some

```
Module EquivDec.
  Class t (A:Type) : Type := Make {
    eq : A → A → Prop ;
    refl : forall x, eq x x ;
    sym : forall x y, eq x y → eq y x ;
    trans : forall x y z, eq x y → eq y z → eq x z ;
    dec : forall x y, Decidable (eq x y) }.
End EquivDec.
Notation "x =♯ y" := (EquivDec.eq x y) (at level 40).
```

(a) Equivalence relation

```
Module PosetDec.
  Class t A : Type := Make {
    eq :> EquivDec.t A ;
    order : A → A → Prop ;
    refl : forall x y,  x =♯ y → order x y ;
    antisym : forall x y, order x y → order y x → x =♯ y ;
    trans : forall x y z, order x y → order y z → order x z ;
    dec : forall x y, Decidable (order x y) }.
End PosetDec.
Notation "x ⊑♯ y" := (PosetDec.order x y) (at level 40).
```

(b) Poset structure

Program 3.2: Coq definition of the decidable poset structure

```
Module TopDec.
  Class t A `{PosetDec.t A} : Type := Make {
    elem : A ;
    prop : forall x : A, x ⊑♯ elem }.
End TopDec.
Notation "⊤♯" := (TopDec.elem) (at level 40).
```

(a) Top element

```
Module BotDec.
  Class t A `{PosetDec.t A} : Type := Make {
    elem : A ;
    prop : forall x : A, elem ⊑♯ x }.
End BotDec.
Notation "⊥♯" := (BotDec.elem) (at level 40).
```

(b) Bottom element

Program 3.3: Coq definition of the classes for $\top$ and $\bot$

parts of the Coq development use the compromise of assuming that the order relation ⊑ is partially decidable: `forall x y, PartiallyDecidable (x ⊑ y)`. Thanks to the type class mechanism, Coq is able to find such an instance from the `PosetDec.t` class.

```
Module Poset.
  Class t A : Type := Make {
    eq :> Equiv.t A ;
    order : A → A → Prop ;
    refl : ∀ x y,  x == y → order x y ;
    antisym : ∀ x y, order x y → order y x → x == y ;
    trans : ∀ x y z, order x y → order y z → order x z }.
End Poset.
Notation "x ⊑ y" := (Poset.order x y) (at level 40).
```

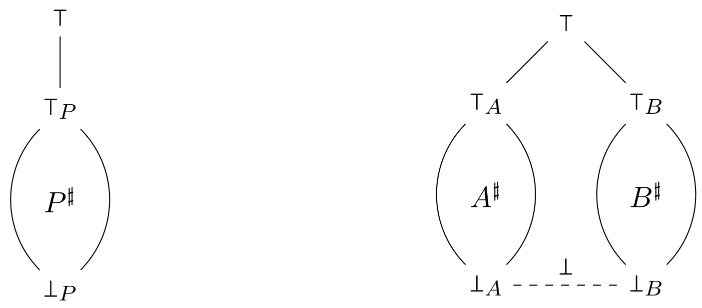
These structures provide all what is needed to understand the Coq definition of a correct concretisation. Type `A` is the abstract domain and Type `C` is the concrete one. The property `gamma_monotone` is the Coq translation of Equation 3.7. Line 3 features the abstract order and Line 4 features the powerset order on concrete sets; this last order is not decidable and thus only uses the ⊑ operator from `Poset.t`. Both orders are inferred by Coq through the type class instances provided in the context.

```
1 Variable gamma : A → C → Prop.
2 Hypothesis gamma_monotone : forall a1 a2,
3   a1 ⊑♯ a2 →
4   gamma a1 ⊑ gamma a2.
```

Interestingly, the type given to the concretisation function `gamma` in Coq is the one of a relation between `A` and `C`. Sets in Coq are indeed represented as predicates: a subset of `C` is of type `C → Prop`. The expected type `A → (C → Prop)` of a function from `A` to a subset of `C` is then exactly the same than this of a relation between `A` and `C`. This representation of concretisation functions as relations is frequent in abstract interpretation when Galois connections are not needed. We shall thus freely consider concretisation functions in this dissertation to be relations when it better suits the intuition.

3.5 Examples of Poset

This section provides some simple examples of generic posets used in this thesis. These posets are used to combine different posets. They are particularly useful to abstract the inputs of extended terms, as these are usually combinations of basic values. Each of these posets has been formalised in Coq.



(a) Adding $\top$ to a poset $P^\sharp$ **(b)** A simple way to abstract the sum of two posets

Figure 3.9: Picturisation of the Hasse diagrams of simple posets

3.5.1 Poset Product

We assume two disjoint concrete sets A and B, abstracted by two posets $A^\sharp$ and $B^\sharp$: they are related by the γ_A and γ_B functions. We assume that each has a greatest element $\top_A$ and $\top_B$, and a smallest element $\bot_A$ and $\bot_B$, that $\gamma_A(\bot_A) = \varnothing$, and that $\gamma_B(\bot_B) = \varnothing$. The hypotheses about the greatest and smallest elements are optional, but they can help the intuition. Let us abstract $A + B$. Note that in contrary to Coq, I consider that A is a subtype of $A + B$: I mainly ignore the constructors `inr` and `inl` in this dissertation.

The intuitive abstraction is to abstract $A + B$ with $(A^\sharp + B^\sharp)^\top/_{\bot_A = \bot_B}$; it is the sum $A^\sharp + B^\sharp$ to which we have added a global $\top$ element, and in which we have merged the two elements $\bot_A$ and $\bot_B$ using a quotient. Adding a global $\top$ element to a poset P is a common operation; we note it $P^\top$; Figure 3.9a pictures how we can complete the Hasse diagram of P to build it. For the smallest value, it would not make sense to add a new $\bot$ smaller than any other value: the concretisations of $\bot_A$ and $\bot_B$ are already empty. It is not a problem for soundness to have several incomparable abstract values with the same concretisation, but it is an issue for preciseness (and for clarity): it is preferable to join both $\bot_A$ and $\bot_B$ into a single value $\bot$. In Coq, this quotient is performed by defining an equivalence relation (instance of `EquivDec.t`) such that $\bot_A$ and $\bot_B$ are equivalent. Figure 3.9b pictures the domain $(A^\sharp + B^\sharp)^\top/_{\bot_A = \bot_B}$.

If the posets $A^\sharp$ and $B^\sharp$ are also lattices, this abstract domain also is: the $\sqcup$ and $\sqcap$ operations naturally follow. The concretisation function of this abstract domain is defined as expected: given $a^\sharp \in A^\sharp$ and $b^\sharp \in B^\sharp$, the concretisation γ is defined as follows.

$$\gamma(\top) = A + B \qquad \gamma\left(a^\sharp\right) = \gamma_A\left(a^\sharp\right) \qquad \gamma\left(b^\sharp\right) = \gamma_B\left(b^\sharp\right)$$

This abstraction works well as long as program variables rarely mix types: a variable which contains elements of A rarely gets elements of B and so on. But if by chance a variable `x` is analysed in both branches of an *if*-condition, such that it is abstracted by $a^\sharp \in$

$A^\sharp \setminus \{\bot_A\}$ in one branch, and by $b^\sharp \in B^\sharp \setminus \{\bot_B\}$ in the other, then we will only get $\top$ as an abstraction for the value of x when exiting the *if*-construct: it is the only abstract value of this domain greater than both $a^\sharp$ and $b^\sharp$. Note that this happens independently of the precision of the domains $A^\sharp$ and $B^\sharp$.

To avoid this, we have to carry both components in parallel: in the previous case, we would carry both values $a^\sharp$ and $b^\sharp$ in the pair $\left(a^\sharp, b^\sharp\right)$. The intuitive meaning of this pair is that if the value of x is in A, then it must be in $\gamma_A\left(a^\sharp\right)$; if it is in B, then it must be in $\gamma_B\left(b^\sharp\right)$. This is exactly what has been done in Section 3.2.2 to abstract expression outputs: concrete expression outputs can either be values or errors. Abstract expression outputs are thus pairs of abstract values and abstract errors in $Val^\sharp \times error^\sharp$. The abstract values of Jensen et al. [JMT09] are an instance of such a domain.

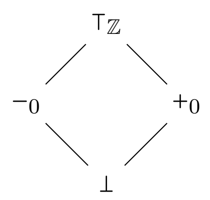
Definition 3.2 (Product poset). We define $(A+B)^\sharp = A^\sharp \times B^\sharp$. We equip it with the following poset structure and concretisation function:

$$\left(a_1^\sharp, b_1^\sharp\right) \sqsubseteq \left(a_2^\sharp, b_2^\sharp\right) \iff a_1^\sharp \sqsubseteq a_2^\sharp \land b_1^\sharp \sqsubseteq b_2^\sharp$$
$$\gamma\left(\left(a^\sharp, b^\sharp\right)\right) = \gamma_A\left(a^\sharp\right) \uplus \gamma_B\left(b^\sharp\right)$$

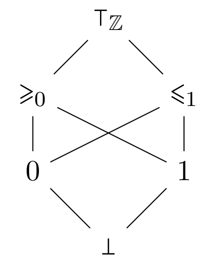
This structure is more precise than the one pictured in Figure 3.9b. Indeed, $\top$ is represented by $(\top_A, \top_B)$, which is the greatest element of $(A+B)^\sharp$, and has the same concretisation $A+B$. Furthermore, any $a^\sharp \in A^\sharp$ can be represented without loss of information (that is, without altering the concretisation function) as $\left(a^\sharp, \bot_B\right)$; the case is similar for any $b^\sharp \in B^\sharp$. For readability purposes, $\left(a^\sharp, \bot_B\right)$ is usually simply written $a^\sharp$, and similarly $\left(\bot_A, b^\sharp\right)$ is usually simply written $b^\sharp$. This notation simplification is also present in Coq thanks to the coercion mechanism (already encountered in Section 2.5.2.1): we shall use this notation in this dissertation whenever it can simplify the understanding. Continuing on this notation, we shall write $\left(a^\sharp, b^\sharp\right)$ as $a^\sharp \sqcup b^\sharp$, which is much simpler to read.

3.5.2 Symbolic Completion of Domains

Abstract interpretation traditionally uses a lot more hypotheses about structures than presented in Section 3.2.4. Our restrictions can lead to frustrating examples in which there is no better element in the abstract structure to abstract a value. For instance, Figure 3.10a represents the Hasse diagram of a subposet of the sign domain (see Figure 3.5a). This specific domain has three abstract values representing the concrete value 0: $-_0$, $+_0$, and $\top$. Among these three abstract values, there is no smallest: $-_0$ is not comparable with $+_0$. This can lead an abstract interpreter to non-deterministic choices when abstracting the concrete value 0: the two choices of $-_0$ and $+_0$ lead to a sound result, but there is no way of knowing in advance which provides the most precise result.



(a) No better abstraction for 0



(b) No smallest greater bound

Figure 3.10: Examples of undesirable posets for abstract interpretation

The abstract domain of Figure 3.10b causes a similar problem. It features the elements $\bot$, 0, 1, $\geqslant_0$, $\leqslant_1$, and $\top_{\mathbb{Z}}$, where 0 and 1 exactly represent the concrete values 0 and 1, $\geqslant_0$ represents the values greater or equals to 0, and $\leqslant_1$ the values lesser or equals to 1. Imagine an *if*-condition assigning to a variable `x` the value 0 in one branch and 1 in the other. To abstract the value of `x` after the conditional, a value greater than both 0 and 1 has to be chosen. There are three such values ($\geqslant_0$, $\leqslant_1$, and $\top_{\mathbb{Z}}$), but no smallest one. Artificially adding a precise $\sqcup$ operator can avoid making such choices.

Definition 3.3. Formally, we start from a poset P and define the symbolic completion $\mathcal{C}(P)$ of P as below. We use the symbols $\vee$ and $\wedge$ to define the extended elements. A similar poset can be defined without the $\wedge$ symbol, which is not always needed. Similarly, we can define a similar poset without the $\vee$ symbol. We do not detail these simpler posets.

$$c \in \mathcal{C}(P) ::= c \vee c \mid c \wedge c \mid e \in P$$

We extend the order of P into $\mathcal{C}(P)$ as follows. If P has a greatest element $\top$, then $\top$ is still the greatest element of $\mathcal{C}(P)$; similarly for a smallest element $\bot$.

$$\begin{aligned}
(c_1 \vee c_2) \sqsubseteq c_3 &\iff c_1 \sqsubseteq c_3 \wedge c_2 \sqsubseteq c_3 \\
(c_1 \wedge c_2) \sqsubseteq c_3 &\iff c_1 \sqsubseteq c_3 \vee c_2 \sqsubseteq c_3 \\
e \sqsubseteq (c_1 \vee c_2) &\iff e \sqsubseteq c_1 \vee e \sqsubseteq c_2 \\
e \sqsubseteq (c_1 \wedge c_2) &\iff e \sqsubseteq c_1 \wedge e \sqsubseteq c_2
\end{aligned}$$

Note that this order is not antisymmetric: we have $\top \vee e \sqsubseteq \top \sqsubseteq \top \vee e$, where $\top$ is (assuming that it exists) the greatest element of P, and e is an element of P. The equivalence relation provided by `EquivDec.t` (see Section 3.4.2) can now be used to quotient $\mathcal{C}(P)$ over the following equivalence relation $\sim$. This operation makes $\sqsubseteq$ antisymmetric.

$$c_1 \sim c_2 \iff c_1 \sqsubseteq c_2 \wedge c_2 \sqsubseteq c_1$$

The order $\sqsubseteq$ has been defined such that the symbols $\vee$ and $\wedge$ define a proper smallest greater bound $\sqcup$ and greatest lower bound $\sqcap$ of the symbolic completion: $\mathcal{C}(P)$ forms a lattice. In CoQ however, the $\sqcup$ and $\sqcap$ operations have been partially optimised to avoid carrying useless terms; for instance, if $c_1 \sqsubseteq c_2$, then $c_1 \sqcup c_2 = c_2$. The operations $\sqcup$ and $\vee$ are nevertheless equivalent with respect to the quotient relation $\sim$.

We now extend the concretisation function γ of P into $\mathcal{C}(P)$. The following definition is compatible with the above order: it respects Equation 3.7.

$$\gamma(c_1 \vee c_2) = \gamma(c_1) \cup \gamma(c_2)$$
$$\gamma(c_1 \wedge c_2) = \gamma(c_1) \cap \gamma(c_2)$$

The symbolic completion domain provides an example of the usage of `PartiallyDecidable` described in Section 3.4.1. We can indeed propagate the partial decidability of the order of P into $\mathcal{C}(P)$. Suppose that $c_1 \sqsubseteq c_2$, but that the `try_to_decide` nevertheless answers `false` to this inequality: the partial decidability procedure failed to prove the order. In such case, we would get $c_1 \sqcup c_2 = (c_1 \vee c_2)$: it is still sound, but not as simple as just c_2 (although equivalent with respect to $\sim$). In this case, the lack of precision of `PartiallyDecidable` only hinders the memory usage of the analysis, not its soundness.

This domain can be used to make domains more precise when needed. In particular, note how given two abstract domains $A^\sharp$ and $B^\sharp$ the abstract domains $\mathcal{C}\left(A^\sharp + B^\sharp\right)$ and $(A + B)^\sharp$ (see Definition 3.2) have the same precision, as they express the same concrete sets. The ability to symbolically complete domains makes the usage of posets instead of lattices less problematic for precision, as it is always possible to complete a domain to be more precise when needed.

3.6 Building an Abstract Semantics

In this chapter, we have mostly covered abstract domains: we discussed their associated hypotheses, and we showed the definition of some of them. Abstract domains are used to abstract the values and memory model of the concrete semantics. They are crucial to define an abstract semantics; but we have not yet described how to define an abstract semantics, and more importantly, how to prove it sound. The fact is that it is usually done in an ad-hoc manner or driven by intuition: the abstract semantics is built after deeply understanding the concrete semantics and its invariants.

Concrete domains do not necessarily provide useful invariants. For instance, JAVASCRIPT's memory model can be abstracted as a heap from locations to objects, objects being maps from fields to values. As-is, there is no invariant claiming that the locations stored in an object of the heap are associated with an object in the same heap. And yet, it is an invariant of JavaScript's semantics. Proving it requires a lot of effort [Lal14]. Furthermore, this invariant has to catch the evolutions of JavaScript's semantics, as it depends on the entire semantics of JavaScript. In a way, proving such invariants amounts to understanding the language semantics. JavaScript's memory model (see Section 1.2.3) is much simpler than JavaScript's whole semantics: Section 1.2.4 presents an example of JavaScript's unnecessary complex semantics.

Here follows a typical rule from the literature. Rule IF-ABS abstracts several rules involved in the *if*-construct at once—the three rules RED-IF, RED-IF-1-POS, and RED-IF-1-NEG. The idea of Rule IF-ABS is to ignore the condition of the *if*-construct, then analyse each branch of the construct, and finally merge the results—supposing that there is a $\sqcup$ operator in the poset of abstract results. It may not be a very precise rule, but we now ignore this matter for the sake of the example.

$$\textsc{if-abs}\quad \frac{E^\sharp, s_1 \Downarrow r_1^\sharp \qquad E^\sharp, s_2 \Downarrow r_2^\sharp}{E^\sharp, \mathit{if}\ (e > 0)\ s_1\ s_2 \Downarrow r_1^\sharp \sqcup r_2^\sharp}$$

The soundness of an abstract semantics is proven globally: once all abstract rules have been defined, we consider the built abstract semantics and prove that it is sound with respect to the concrete semantics. This induction can be huge, and it may require some work to find a bug in the abstract semantics during the process. Intuition is often not enough in such contexts: however natural Rule IF-ABS may appear, it is not sound. Indeed, there is a concrete rule which can apply at an *if*-construct, but which is not taken into account in this abstract rule: Rule RED-ERROR-STAT, which is triggered when the evaluation of the expression e returns an error. The concrete derivation below is missed by the abstract rule IF-ABS: the abstract semantics may state that no error can happen and miss the result err. The abstract rule IF-ABS is thus not sound.

$$\frac{\textsc{red-var-undef}\ \dfrac{}{\epsilon, \mathsf{x} \Downarrow err} \qquad \dfrac{}{\epsilon, err, \mathit{if}_1\ \mathit{skip}\ \mathit{skip} \Downarrow err}\ \textsc{red-error-stat}}{\epsilon, \mathit{if}\ (\mathsf{x} > 0)\ \mathit{skip}\ \mathit{skip} \Downarrow err}\ \textsc{red-if}$$

This mistake was easy to catch in the small language which we considered, given its small number of rules, but what about JavaScript? JavaScript can be quite complex, and having a correct intuition about how it works and how to abstract the 900 rules of JSCert is challenging. We approach this issue by proposing a new method of abstracting semantics. This method is a continuation of Schmidt's work about the abstraction of big-step semantics (see Section 3.3). The overall semantics of JavaScript is complex, but each rule of JSCert is simple. The method presented in Chapter 4 aims at *independently* abstract each concrete rule. There is thus no need to understand how the semantic behaves in a global scale, but only to understand how the memory model works. In this setting, a rule such as Rule IF-ABS is no longer needed.

4 Principles for Building Analysers of Large Semantics

> *Or would Professor McGonagall have given it to him anyway, only later in the day, whenever he got around to asking about his sleep disorder or telling her about the Sorting Hat's message? And would he, at that time, have wanted to pull a prank on himself which would have led to him getting the Time-Turner earlier? So that the only self-consistent possibility was the one in which the Prank started before he even woke up in the morning...?*
>
> Eliezer Yudkowsky [Yud15]

We have seen in Chapter 1 how complex the JavaScript language is. Building certified analyses for the full language thus appears to be a difficult task. At first sight, we would like to avoid supporting JavaScript's full semantics and to restrict ourselves to a safe sublanguage. This is at what libraries such as ADSafe [Cro08] aim. But how can we prove that these libraries safely restrict to a sublanguage when `eval`-like features hide within the constructor of functions (see Section 1.2.5)? Dually, programmers usually use a large part of the language peculiarities: choosing what should be considered a programming error and what should not is a difficult task by itself. An alternative to these libraries would consist in building a certified JavaScript analyser for the full JavaScript language.

The proof effort needed by the framework of abstract interpretation grows with the size of definitions: in the case of JavaScript, the proof effort is overwhelming. The $\mathbb{K}$ framework is able to generate analysers from a concrete semantics and some abstract domains—there are no formally proven guarantees on these analysers, though. This idea of generating analysers from a concrete semantics appears to be a good solution to ease the Coq development. This chapter presents some techniques to extend the abstract interpretation framework briefly presented in Chapter 3 to large semantics. To generate an abstract semantics from a concrete one, we need to have a definition of what a semantics is—in other words, we need semantics to be first-class citizens of Coq. This chapter presents how to define and prove sound such an abstract semantics. Notably, this chapter does not focus on proving abstract interpreters sound (although Section 4.6 briefly mentions how to certify some of them), but on proving abstract semantics sound. The work presented in this chapter resulted in a publication [BJS15b] as well as a Coq development [BJS15a].

$$
\begin{aligned}
e &::= c \in \mathbb{Z} \\
&\mid \mathsf{x} \in \mathit{Var} \\
&\mid e_1 + e_2
\end{aligned}
\qquad
\begin{aligned}
e_e &::= \cdot +_1 e \\
&\mid \cdot +_2 \cdot
\end{aligned}
\qquad
\begin{aligned}
s &::= \mathit{skip} \\
&\mid s_1 ; s_2 \\
&\mid \mathsf{x} := e \\
&\mid \mathit{if}\ (e > 0)\ s_1\ s_2 \\
&\mid \mathit{while}\ (e > 0)\ s
\end{aligned}
\qquad
\begin{aligned}
s_e &::= \mathsf{x} :=_1 \cdot \\
&\mid \cdot ;_1 s_2 \\
&\mid \mathit{if}_1\ s_1\ s_2 \\
&\mid \mathit{while}_1\ (e > 0)\ s \\
&\mid \mathit{while}_2\ (e > 0)\ s
\end{aligned}
$$

Figure 4.1: Updating the language of Figure 3.3

$$
\textsc{red-while}\quad \frac{E, e \Downarrow r \qquad E, r, \mathit{while}_1\ (e > 0)\ s \Downarrow r'}{E, \mathit{while}\ (e > 0)\ s \Downarrow r'}
\qquad
\textsc{red-while-1-neg}\quad \frac{}{E, v, \mathit{while}_1\ (e > 0)\ s \Downarrow E}\ v \leqslant 0
$$

$$
\textsc{red-while-1-pos}\quad \frac{E, s \Downarrow r \qquad r, \mathit{while}_2\ (e > 0)\ s \Downarrow r'}{E, v, \mathit{while}_1\ (e > 0)\ s \Downarrow r'}\ v > 0
\qquad
\textsc{red-while-2}\quad \frac{E, \mathit{while}\ (e > 0)\ s \Downarrow r}{E, \mathit{while}_2\ (e > 0)\ s \Downarrow r}
$$

Figure 4.2: Rules for the *while*-construct

Section 4.2 presents traditional abstract rules found in the literature. These rules are very different from the corresponding concrete rules, making their soundness difficult to prove. The pretty-big-step format proved to be a good basis to build and define abstract semantics. Section 4.3 presents a formalisation of this semantic style. Section 4.4 then presents how abstract semantics can be defined and proven in this setting. Section 4.6 presents how generic abstract interpreters can be developed.

4.1 Language and Domain

We use in this chapter the same language as the one presented in Figures 3.3 and 3.4, but with an additional looping construct: $\mathit{while}\ (e > 0)\ s$. As for the *if*-construct, the "> 0" is part of the syntax of the *while*-construct. The rules associated with this new construct are shown in Figure 4.2. Rule RED-WHILE computes the expression, then Rules RED-WHILE-1-POS and RED-WHILE-1-NEG check the result of this computation; in the positive case, the statement s is executed and Rule RED-WHILE-2 checks that its result does not abort. This language is not a particularly large semantics: this chapter presents a technique which scales with the size of the semantics, using this language as an example. Section 4.7.1 then extends this language and evaluates how scalable this technique is. This technique aims to be applied to the JSCERT formalisation on further works.

The techniques presented in this chapter, as well as the COQ formalisation, are parametrised by domains: they can be applied to any concrete domain, abstract poset, and concretisation function γ. The examples of this chapter use a simple abstract domain, following

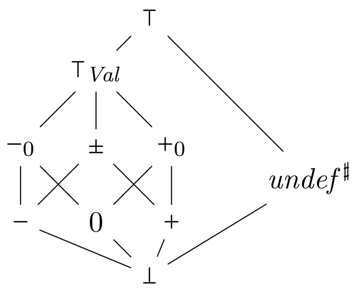


Figure 4.3: The Hasse diagram of the $Store^\sharp$ poset

what has been defined in Section 3.2.2. The goal of the example analyses shall be to check whether a program can result in err at the end of its execution. The concrete semantics of Figure 3.4 is made so that this happens when an undefined variable is read.

We first abstract each construct of Definition 3.1. The domain of integers is abstracted by the abstract domain of signs. The singleton domain of errors is abstracted by the two-point domain $\left\{err^\sharp, \overline{err}^\sharp\right\}$, ordered such that $\overline{err}^\sharp \sqsubseteq err^\sharp$. The absence of errors is represented by $\overline{err}^\sharp$, and the possible presence of an error by $err^\sharp$. The result of an expression is either a value or an error in Out_e; we abstract this sum domain by the poset product of Definition 3.2. A result known to be an error is thus abstracted by $\left(\bot, err^\sharp\right) \in Out_e^\sharp$. As explained in Section 3.5.1, we will simply write this abstract output $err^\sharp$. Similarly, we consider that abstract values can be coerced into abstract expression outputs.

Environments are more complex as their domains change during execution time. We consider them as total functions from variables to special abstract values in $Store^\sharp$; these special values represent either values or undefined variables. We could use the poset product to represent this sum type, but we shall keep these values simple for now and use a sum type in the abstract domain. Figure 4.3 shows the Hasse diagram of the $Store^\sharp$ poset. If a variable `x` is mapped by an abstract environment $E^\sharp$ to the top element $\top_{Env^\sharp}$ of the $Store^\sharp$ poset, then $E^\sharp$ provides no information about whether `x` is defined. Environments are ordered point-wise; their greatest element $\top_{Env^\sharp}$ and their smallest element $\bot_{Env^\sharp}$ respectively maps every variable into $\top_{Store^\sharp}$ and $\bot_{Store^\sharp}$. From environment, we can define statement outputs as the poset product of environments and errors.

The full abstract domains are shown below. They mirror the concrete domains of Definition 3.1. These domains—as well as the rest of this chapter—have been implemented in Coq [BJS15a]: the Coq name of these construct have also been indicated. Incidentally, the poset of the abstract domains are also complete lattices. This last property is not required by the framework presented in this chapter, but it can help generate more precise results. It is good practise to choose complete lattices as abstract domains when possible.

- $Val^\sharp = Sign = \{\bot, -, 0, +, -_0, \pm, +_0, \top_\mathbb{Z}\}$, named `aVal` in the Coq files;
- $error^\sharp = \left\{\overline{err}^\sharp, err^\sharp\right\}$, `aErr` in Coq;
- $Store^\sharp = \left(Val^\sharp + undef^\sharp\right)^\top$;

```
Inductive ares : Type :=
  | ares_expr : aOute → ares
  | ares_prog : aOuts → ares
  | ares_top : ares
  | ares_bot : ares.
```

(a) Abstract results

```
Inductive ast : Type :=
  | ast_expr : aEnv → ast
  | ast_stat : aEnv → ast

  | ast_add_1 : aEnv → sign_ares → ast
  | ast_add_2 : aVal → sign_ares → ast

  | ast_asgn_1 : aEnv → sign_ares → ast
  | ast_seq_1 : sign_ares → ast
  | ast_if_1 : aEnv → sign_ares → ast
  | ast_while_1 : aEnv → sign_ares → ast
  | ast_while_2 : sign_ares → ast

  | ast_top : ast
  | ast_bot : ast.
```

(b) Abstract semantic contexts

Program 4.1: Coq definition of the abstract results and semantic contexts

- $Env^\sharp = Var \to Store^\sharp$, `aEnv` in Coq;
- $Out_e^\sharp = Val^\sharp \times error^\sharp$, `aOute` in Coq;
- $Out_s^\sharp = Env^\sharp \times error^\sharp$, `aOuts` in Coq.

There are places in this chapter in which the two kinds of outputs $Out_e^\sharp$ and $Out_s^\sharp$ have to be merged. These results are implemented as a sum augmented with a top element $\left(Out_e^\sharp + Out_s^\sharp\right)^\top / \bot_{Out_e^\sharp} = \bot_{Out_s^\sharp}$. The new top element indicates a type error due to a confusion of expressions and statements during the analysis. The two abstract values with empty concretisations, $\bot_{Out_e^\sharp} = \left(\bot_{Sign}, \overline{err}^\sharp\right)$ and $\bot_{Out_s^\sharp} = \left(\bot_{Env^\sharp}, \overline{err}^\sharp\right)$, have been merged into a single $\bot$ element. Program 4.1a shows the Coq definition of the abstract result type $res^\sharp$. Similarly to results, semantic contexts are abstracted as a sum $st^\sharp$. In contrary to the rules of Figures 3.4 and 4.2, we chose here to distinguish the different semantic contexts for each extended terms. This is more a design choice than a real constraint, but it helps catching early errors when defining semantics. Program 4.1b shows their Coq definition[1].

When building abstract semantics, we sometimes have to check whether an abstract result contains the concrete value err in its concretisation. We can take advantage of the order of the product poset: given $r^\sharp \in Out_e^\sharp$, having $err \in \gamma\left(r^\sharp\right)$ implies that $\left(\bot_{Val^\sharp}, err^\sharp\right) \sqsubseteq r^\sharp$.

[1] Due to an unfortunate name conflict, the Coq type `ast` spells like the type of abstract syntax tree (AST). It however represents an abstract semantic context, whose concrete version is written `st` in Coq.

Using the coercions of the product poset, this last ordering is written $err^\sharp \sqsubseteq r^\sharp$. The other way—that $err^\sharp \sqsubseteq r^\sharp$ implies $err \in \gamma\left(r^\sharp\right)$—is also true, but not needed for the soundness of our analyses. It is needed to prove completeness, though, but we are not considering this case. Similarly, given $r^\sharp \in Out_s^\sharp$, having $err \in \gamma\left(r^\sharp\right)$ implies $err^\sharp \sqsubseteq r^\sharp$ (the converse being true but not needed). We are here abusing notations, considering $err^\sharp$ to be in $error$, $Out_e^\sharp$, or $Out_s^\sharp$ depending on the context. Overall, given a result $r^\sharp \in \left(Out_e^\sharp + Out_s^\sharp\right)_\perp^\top$, having $err \in \gamma\left(r^\sharp\right)$ implies $\left(\perp_{Val^\sharp}, err^\sharp\right) \sqsubseteq r^\sharp$ or $\left(\perp_{Env^\sharp}, err^\sharp\right) \sqsubseteq r^\sharp$, which we will simply write $err^\sharp \sqsubseteq r^\sharp$ by abuse of notations. Again, the converse is true but not needed for soundness. We can now define the error projection $r^\sharp\big|_{error}$ of the abstract result $r^\sharp \in \left(Out_e^\sharp + Out_s^\sharp\right)_\perp^\top$: if $err^\sharp \sqsubseteq r^\sharp$ (and thus $err \in \gamma\left(r^\sharp\right)$), then $r^\sharp\big|_{error} = err^\sharp$, otherwise $r^\sharp\big|_{error} = \perp$. Similarly, it is sometimes needed to get the projection of a result into a value (for Out_e) or an environment (for Out_s). Given $r^\sharp = \left(v^\sharp, er\right) \in Out_e$, we write $r^\sharp\big|_{Val^\sharp}$ the value part $v^\sharp$ of this result. We extend this notation into $r^\sharp \in \left(Out_e^\sharp + Out_s^\sharp\right)_\perp^\top$ by defining $\top\big|_{Val^\sharp} = \top_{\mathbb{Z}}$ and $r^\sharp\big|_{Val^\sharp} = \perp$ for $r^\sharp = \perp$ or $r^\sharp \in Out_s$. We define similarly $r^\sharp\big|_{Env^\sharp}$ as being the abstract environment of a result $r^\sharp \in Out_s$, generalised into $r^\sharp \in \left(Out_e^\sharp + Out_s^\sharp\right)_\perp^\top$ with $\top\big|_{Env^\sharp} = \top_{Env^\sharp}$ and $r^\sharp\big|_{Env^\sharp} = \perp$ for other kinds of results ($\perp$ or $r^\sharp \in Out_e$).

4.2 Traditional Abstract Rules

In Section 3.3, we have seen how Schmidt formalised concrete and abstract derivations, as well as their relation. The approach of this chapter is similar: concrete and abstract executions are assemblages of *rules*. We concluded the previous chapter by stating that most abstract semantics were built in ad-hoc ways. The goal of this chapter is to provide principles to derive these abstract rules whilst limiting the associated proof effort. This section presents some traditional abstract rules and shows how different they are from the concrete semantics of Figures 3.4 and 4.2. In order to minimise the proof effort, this chapter proposes to build the abstract semantics as close as possible to the concrete semantics.

Figure 4.4 shows some examples of abstract rules found in the literature. Rule IF-ABS-CORRECTED is a corrected version of the unsound Rule IF-ABS of Section 3.6. As $\perp$ is neutral for $\sqcup$, Rule IF-ABS-CORRECTED propagates the potential aborting result of the expression evaluation. Rule IF-ABS-REFINED represents a more precise version of this rule found in some analysers [Jou+15]: each branch is restricted to the semantic contexts which can trigger this branch. The notation $E^\sharp\big|_{e>0}$ aims at giving an intuition of what is happening, but is not meant to be formal. Such refining operations depend on the chosen domains and are common in symbolic analyses [Min06a].

Rule WHILE-ABS-FIXED-POINT only applies if the expression only returns non-aborting results. The condition $E^\sharp, e \Downarrow v^\sharp$ could be rewritten $E^\sharp, e \Downarrow r^\sharp \wedge err^\sharp \not\sqsubseteq r^\sharp$. This rule requires to find an error-free fixed point $E^\sharp$ of the analysed loop. This rule is sound: it requires that no error occur during the execution, which ensures that only the rules of Figure 4.2 apply. Then, as by hypothesis the statement s leaves the abstract state $E^\sharp$ unchanged, $E^\sharp$

$$\textsc{if-abs-corrected}\quad \dfrac{E^\sharp, e \Downarrow r_0^\sharp \qquad E^\sharp, s_1 \Downarrow r_1^\sharp \qquad E^\sharp, s_2 \Downarrow r_2^\sharp}{E^\sharp, \mathit{if}\ (e > 0)\ s_1\ s_2 \Downarrow r_0^\sharp\big|_{\mathit{error}} \sqcup r_1^\sharp \sqcup r_2^\sharp}$$

$$\textsc{if-abs-refined}\quad \dfrac{E^\sharp, e \Downarrow r_0^\sharp \qquad E^\sharp\big|_{e>0}, s_1 \Downarrow r_1^\sharp \qquad E^\sharp\big|_{e\leqslant 0}, s_2 \Downarrow r_2^\sharp}{E^\sharp, \mathit{if}\ (e > 0)\ s_1\ s_2 \Downarrow r_0^\sharp\big|_{\mathit{error}} \sqcup r_1^\sharp \sqcup r_2^\sharp}$$

$$\textsc{while-abs-fixed-point}\quad \dfrac{E^\sharp, e \Downarrow v^\sharp \qquad E^\sharp, s \Downarrow E^\sharp}{E^\sharp, \mathit{while}\ (e > 0)\ s \Downarrow E^\sharp}$$

$$\textsc{while-abs-precise-fixed-point}\quad \dfrac{E^\sharp, e \Downarrow v^\sharp \qquad E^\sharp\big|_{e>0}, s \Downarrow E^\sharp}{E^\sharp, \mathit{while}\ (e > 0)\ s \Downarrow E^\sharp\big|_{e\leqslant 0}}$$

$$\textsc{abs-top}\quad \dfrac{}{\sigma^\sharp, t \Downarrow \top}$$

$$\textsc{abs-weaken}\quad \dfrac{\sigma_1^\sharp \sqsubseteq \sigma_2^\sharp \qquad \sigma_2^\sharp, t \Downarrow r_2^\sharp \qquad r_2^\sharp \sqsubseteq r_1^\sharp}{\sigma_1^\sharp, t \Downarrow r_1^\sharp}$$

Figure 4.4: Examples of abstract rules used in abstract semantics

is indeed an invariant of the loop. Similarly to Rule IF-ABS-REFINED, the environment $E^\sharp$ can be refined when executing the statement and leaving the loop, as in Rule WHILE-ABS-PRECISE-FIXED-POINT. These two rules about the *while*-construct only consider terminating results: if the program terminates, then its result will be abstracted by $E^\sharp$ (or $E^\sharp\big|_{e\leqslant 0}$), but no guarantee is given that the program will terminate. We shall not consider non-terminating behaviours in this dissertation. Focussing only on terminating behaviours enables us to define Rule ABS-TOP, which returns $\top$ on every terms and semantic contexts (supposing that there exists a greatest abstract result $\top$): any concrete result is supposed to be in the concretisation of the $\top$ result, thus if a concrete derivation terminates on a term, then the result can be abstracted by $\top$. Rule ABS-TOP, but also the two rules about the *while*-construct, reduce whole derivations of unknown size into a single rule.

Rule WHILE-ABS-FIXED-POINT is difficult to use: in most cases, running a statement s on an abstract state $\sigma^\sharp$ returns a different abstract state. To apply Rule WHILE-ABS-FIXED-POINT, another abstract rule is needed to be able to "tie the knot". To this end, Rule ABS-WEAKEN enables us to loose precision on the resulting abstract state. Inferring a loop invariant from a given *while*-construct is a complex but orthogonal task [CC92]: we are only interested in proving that such an invariant is correct once computed by another method.

Most of the rules of Figure 4.4 abstract several rules at once, making the proof of their soundness—or to even self convince that they are sound—difficult tasks. Rules ABS-TOP and WHILE-ABS-FIXED-POINT push this concept further by abstracting whole derivations. The abstract and the concrete semantics have thus little in common, and the proof of soundness of the abstract semantics has to be done on a case-by-case basis. We propose a different approach based on the pretty-big-step format to build abstract semantics.

4.3 Formalising the Pretty-big-step Format

In Section 2.5.2.1, the pretty-big-step style was used because it fits the writing style of ECMAScript. This chapter shows another interesting property of pretty-big-step: its constraints greatly facilitate its formalisation. Big-step operational semantics are inductively defined with rules (or, more precisely, rule schemes) of the following form, describing how a term t evaluates in a semantic context σ of type st to a result r of type res.

$$\dfrac{\sigma_1, t_1 \Downarrow r_1 \qquad \sigma_2, t_2 \Downarrow r_2 \qquad \dots}{\sigma, t \Downarrow r}\ \textsc{Name} \quad \textit{side-conditions}$$

There are several implicit relations between the elements of such rule schemes which need to be made explicit in order to provide a functional representation of derivation rules. The pretty-big-step format helps making these implicit relations explicit.

4.3.1 Definition of Rules

Pretty-big-step adds constraints on rule shapes. In particular, it makes explicit the components of rules: rule names, side-conditions, rule structure, and transfer functions. We now describe each of these components.

First, the name of a rule should completely identify it. The rules of Figures 3.4 and 4.2 are actually rule schemes and do not respect this constraint. In particular, the term on which a rule applies should be inferable from its name (also called identifier). To this end, we complete the rule names with the needed information. For instance, the rule accessing the variable x is named by RED-VAR(x). Some examples of updated rules are shown below. The full updated semantics can be found in the webpage accompanying this thesis [Bod16].

$$\dfrac{}{E, \mathsf{x} \Downarrow E[\mathsf{x}]}\ \textsc{red-var}(\mathsf{x}) \quad \mathsf{x} \in dom(E) \qquad \dfrac{E, e_1 \Downarrow r \qquad E, r, \cdot +_1 e_2 \Downarrow r'}{E, e_1 + e_2 \Downarrow r'}\ \textsc{red-add}(e_1, e_2)$$

$$\dfrac{E, e_2 \Downarrow r \qquad E, v_1, r, \cdot +_2 \cdot \Downarrow r'}{E, v_1, \cdot +_1 e_2 \Downarrow r'}\ \textsc{red-add-1}(e_2) \qquad \dfrac{}{E, v_1, v_2, \cdot +_2 \cdot \Downarrow v_1 + v_2}\ \textsc{red-add-2}$$

Formally, a pretty-big-step semantics carries a set $\mathcal{N}$ of rule names and a function $rule : \mathcal{N} \to Rule$ mapping rule names to actual rules (the type $Rule$ is described below). They also provide a function $\mathfrak{l} : \mathcal{N} \to term$ (standing for "left") mapping rule names to the term to which the rule applies. For instance, for Rule RED-VAR(x), we have $\mathfrak{l}_{\textsc{red-var}(\mathsf{x})} = \mathsf{x}$.

Second, rules have *side-conditions*. We impose a clear separation between these conditions and the continuation of the derivation above the inference line. The conditions involve the rule name $\mathfrak{r}$ and the semantic context σ; they are expressed as a predicate $cond : \mathcal{N} \to st \to Prop$ which states whether Rule $\mathfrak{r}$ applies in the given context σ. For instance, two rules can apply to the term x (a variable), depending on whether this variable is defined in the given environment E: it is either the look-up rule RED-VAR($\mathtt{x}$) or the error rule RED-VAR-UNDEF($\mathtt{x}$). This gives the following *cond* predicates.

$$
\begin{aligned}
\mathit{cond}_{\textsc{red-var}(\mathtt{x})}(E) &= \mathtt{x} \in \mathit{dom}(E) \\
\mathit{cond}_{\textsc{red-var-undef}(\mathtt{x})}(E) &= \mathtt{x} \notin \mathit{dom}(E)
\end{aligned}
$$

Third, in contrary to big-step which accepts any number of premises above the inference line, the pretty-big-step format restricts this number to at most two. There are thus three possible rule formats in pretty-big-step: axioms (with no premise), rules with one inductive premise, and rules with two, respectively written Ax, R_1, and R_2. The function $\mathit{kind} : \mathcal{N} \to \{Ax, R_1, R_2\}$ returns the format (axiom, rule 1, or rule 2) of a rule.

The rule itself is described by the *Rule* type. This type contains two kinds of information: the syntactic and the semantic aspects of the rule. Indeed, to evaluate a rule, one needs to specify which terms to inductively consider (syntactic aspects) and how the semantic contexts and results are propagated (semantic aspects). We first describe the former.

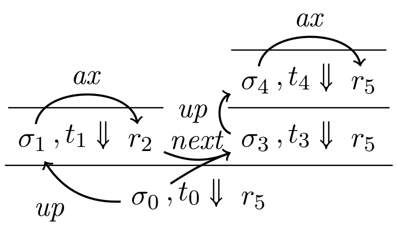
Axioms have no inductive premises, thus carrying no additional terms. In format 1 rules (rules with one hypothesis), the current computation is redirected to the computation of the semantics of another term (often a sub-term). Rule RED-IF-1-POS(s_1, s_2) is a typical instance: it redirects the computation to the term s_1. The syntactic aspect of format 1 rules contains a term $\mathfrak{u}_1$ (standing for "up"). Similarly, format 2 rules have two inductive premises, and thus carry two terms $\mathfrak{u}_2$ and $\mathfrak{n}_2$ (standing for "next"). These syntactical information are implemented in Coq with the type `Rule_struct` defined below.

```
Inductive Rule_struct (term : Type) : Type :=
  | Rule_struct_Ax : Rule_struct term
  | Rule_struct_R1 : term →  Rule_struct term
  | Rule_struct_R2 : term → term → Rule_struct term.
```

The functions *kind*, $\mathfrak{l}$, as well as the syntactic aspects of rules $\mathfrak{u}_1$, $\mathfrak{u}_2$, and $\mathfrak{n}_2$ describe the *structure* of a rule. It is defined in Coq as follows ($\mathfrak{l}$ is named `left` in the development).

```
Record structure := {
    term : Type ;
    name : Type ;
    left : name → term ;
    rule_struct : name → Rule_struct term }.
```

The structure of a rule provides a lot of information about how this rule can be assembled with other rules. It does not provide any information about how the rule manipulates semantic contexts and results. This computation is done in the transfer functions, also contained in the constructions of type $Rule$. Transfer functions also depend on the format of their rule. They can be summed up in the following informal scheme, detailed below.



Axioms directly return a result. They carry one (partial) transfer function $ax : st \rightharpoonup res$. Every context does not trigger the rule because of the $cond$ predicate. The transfer function might make no sense in other contexts: what Rule RED-VAR(x) is supposed to return if x is not in the domain of the current environment? This is the reason why transfer functions are partial functions. However, if the rule $\mathfrak{r}$ is an axiom with the transfer function ax and σ a semantic context such that $cond_{\mathfrak{r}}(\sigma)$, then $ax(\sigma)$ should intuitively be defined. This property is called the *exhaustivity* of a rule, and is discussed in Section 4.4.4.

Format 1 rules have to compute a new semantic context, which will be passed to their premise. They are thus associated with a transfer function $up : st \rightharpoonup st$. As for axioms, the up function is partial. Once the premise finishes its computation to a result r, this result is directly propagated (see Figure 4.5). The result of the premise can not be changed by the format 1 rule: there is no $down$ transfer function. This constraint can be put in parallel to the monadic style of JSRef (see Section 2.6.2): once the local computation has been performed, the computation is entirely transferred to the continuation. Thanks to this constraint, a rule can not "refuse" a result given by an inductive premise by having an undefined $down$ function on this result, as the big-step Rule EXEC-APP of Figure 2.1b which forces the expression e_1 to result in a λ-abstraction. This constraint guarantees that the applicability of a rule only depends on local conditions specified by $\mathfrak{l}$ and $cond$.

Format 2 rules start similarly than format 1, with a transfer function $up : st \rightharpoonup res$. The computation is then transferred to the first premise. The result of this first branch is then passed to a transfer function $next : st \rightharpoonup res \to st$ along with the initial semantic context. This second transfer function merges the two states. Rule RED-ADD(e_1, e_2) illustrate this process: once the expression e_1 computed a result (hopefully a value v), this result is packaged with the environment E for the continuation so that the expression e_2 can also evaluate. The computation then proceeds to the second premise, whose result is propagates as-is (see Figure 4.5). Importantly, the $next$ transfer function is partial for the semantic context σ (for the same reasons than up), but total for the result r: the $next$ transfer function can not refuse a result once computed, as for format 1 rules. This constraint is the

$$\frac{\mathfrak{r}}{\sigma, \mathfrak{l}_{\mathfrak{r}} \Downarrow ax\left(\sigma\right)} \quad cond_{\mathfrak{r}}\left(\sigma\right) \qquad\qquad \frac{\mathfrak{r} \quad up\left(\sigma\right), \mathfrak{u}_{1,\mathfrak{r}} \Downarrow r}{\sigma, \mathfrak{l}_{\mathfrak{r}} \Downarrow r} \quad cond_{\mathfrak{r}}\left(\sigma\right)$$

$$\frac{\mathfrak{r} \quad up\left(\sigma\right), \mathfrak{u}_{2,\mathfrak{r}} \Downarrow r \qquad next\left(\sigma, r\right), \mathfrak{n}_{2,\mathfrak{r}} \Downarrow r'}{\sigma, \mathfrak{l}_{\mathfrak{r}} \Downarrow r'} \quad cond_{\mathfrak{r}}\left(\sigma\right)$$

Figure 4.5: Rule formats

reason why Program 4.1b contains semantic contexts such as `ast_add_1` carrying results of type `sign_ares`: as the computation of an expression can only return a result of type `aOute`, we can be tempted to define `ast_add_1` as carrying an `aOute` and not a general result; but the *next* transfer functions have to consider all possible results, forcing the semantic contexts to be more general.

Figure 4.5 summarizes the three kinds of rule formats of pretty-big-step. Note that transfer functions only depend on the intermediary semantic states, as well as indirectly on the rule name (which is the only argument of the function *rule*, which provides the transfer functions). In particular, transfer functions do not depend on the currently evaluated term: the *ax* transfer function of Rule RED-ASN-1(x) “knows” that the variable to be assigned is x because x is present in the rule name. This is the reason why we requested rule names to be precise: working with rules is easier than working with rule schemes.

The *cond* predicate and the transfer functions form the semantic aspect of rules. We define them in Coq as follows. The types *st* and *res* are also included into this structure.

```
Record semantics := make_semantics {
    st : Type ;
    res : Type ;
    cond : name → st → Prop ;
    rule : name → Rule st res }.
```

The Coq type `Rule` (parametrised by the types of semantic contexts and results) contains the transfer functions of the rule; it is defined as below. As explained in Section 2.1.2, every Coq function is total: partial functions are implemented with the `option` type. They return `None` if the rule does not apply, either because the semantic context does not have the correct shape, or if the condition to apply the rule is not satisfied. The *next* transfer function has been given the type `st → res → option st` instead of the expected `st → option (res → st)`. This formalisation choice was made to ease the definition of semantics; Section 4.4.4 explains why this formalisation choice does not cause any issue.

```
Inductive Rule st res :=
  | Rule_Ax : (st → option res) → Rule st res
  | Rule_R1 : (st → option st) → Rule st res
  | Rule_R2 : (st → option st) → (st → res → option st) → Rule st res.
```

4

It may seem that we compute the same thing twice: $cond_{\mathfrak{r}}(\sigma)$ states that Rule $\mathfrak{r}$ applies to σ, whilst ax (or the corresponding transfer function) should also return `None` if the rule can not be applied. This second requirement is relaxed to enable simpler definitions: transfer functions may return a result even if they do not apply. For instance, the ax transfer function of Rule RED-VAR-UNDEF$(\mathtt{x})$ always returns `Some` (err); but it may only be applied if the variable $\mathtt{x}$ is not in the environment. This separation between side-conditions and transfer functions is a separation between the control flow and the actual computation. In the Coq development, the first one is implemented using predicates, and the second using computable functions.

All the functions described above, both syntactical and semantic, can be inferred from the rules of Figures 3.4 and 4.2, and conversely. Consider for instance Rule RED-ADD-1(e_2).

$$\textsc{red-add-1}(e_2)\quad \frac{E, e_2 \Downarrow r \qquad E, v_1, r, \cdot +_2 \cdot \Downarrow r'}{E, v_1, \cdot +_1 e_2 \Downarrow r'}$$

This rule is a format 2 rule applying on term $\mathfrak{l}_{\textsc{red-add-1}(e_2)} = \cdot +_1 e_2$. Its structure is defined by the two terms $\mathfrak{u}_{2,\textsc{red-add-1}(e_2)} = e_2$ and $\mathfrak{n}_{2,\textsc{red-add-1}(e_2)} = \cdot +_2 \cdot$. Its $cond$ predicate is implicitly given by notations: it only accepts semantic contexts of the form (E, v), where E is an environment and v a value. In particular, if the term e_1 results in an error in Rule RED-ADD(e_1, e_2), then Rule RED-ADD-1(e_2) will not apply (but Rule RED-ERROR-EXPR$(\cdot +_1 e_2)$ will). We thus have $cond_{\textsc{red-add-1}(e_2)}(\sigma) = \exists E \in Env, v \in Val.\ \sigma = (E, v)$. Its up transfer function removes the v_1 part of the semantic context, and its $next$ transfer function combines the newly computed result r with the old semantic context:

$$rule\,(\textsc{red-add-1}\,(e_2)) = R_2\,(\lambda\,(E, \_)\,.\,E, \lambda\,(E, v)\,r.\ (E, v, r))$$

By decomposing each rule into several functional components, we have defined a data structure for pretty-big-step semantics. Importantly, this structure is divided into two aspects: the syntactic aspects (the rule names and the various terms which they carry), and the semantic aspects (side-conditions and transfer functions). We now describe how to assemble rules to build a concrete derivation.

4.3.2 Concrete Semantics

The concrete semantics is given in the form of an evaluation relation (or equivalently, a set of semantic triples) $\Downarrow \in \mathcal{P}(st \times term \times res)$. Semantic triples relate semantic contexts σ and terms t to their result(s) r, they are naturally written $\sigma, t \Downarrow r$. The predicate $\Downarrow$ is defined as a fixed point of the *immediate consequence* operator $\mathcal{F}$, which we now detail.

$$\mathcal{F} : \mathcal{P}(st \times term \times res) \to \mathcal{P}(st \times term \times res)$$

Let $\Downarrow_0 \in \mathcal{P}(st \times term \times res)$ be an evaluation relation. The immediate consequence $\mathcal{F}$ proceeds in two steps: it selects a rule which applies, then applies it. We first describe the second step: the rule application. The application relation for the rule $\mathfrak{r}$, written $apply_{\mathfrak{r}}(\Downarrow_0) : \mathcal{P}(st \times term \times res)$ proceeds as follows. It accepts a semantic triple (σ, t, r) if it can be computed with the rule $\mathfrak{r}$ using the premises given by $\Downarrow_0$. This function is thus based on the transfer functions of the rule $\mathfrak{r}$, as shown below. Note how the relation $\Downarrow_0$ is ignored when the rule $\mathfrak{r}$ is an axiom.

$$
\begin{array}{l}
apply_{\mathfrak{r}}(\Downarrow_0) := \\
\quad match\ rule(\mathfrak{r})\ with \\
\quad \mid\ Ax(ax) \Rightarrow \{(\sigma, \mathfrak{l}_{\mathfrak{r}}, r) \mid ax(\sigma) = \mathtt{Some}(r)\} \\
\quad \mid\ R_1(up) \Rightarrow \left\{(\sigma, \mathfrak{l}_{\mathfrak{r}}, r) \;\middle|\; \begin{array}{l} up(\sigma) = \mathtt{Some}(\sigma') \\ \wedge\, \sigma', \mathfrak{u}_{1,\mathfrak{r}} \Downarrow_0 r \end{array}\right\} \\
\quad \mid\ R_2(up, next) \Rightarrow \left\{(\sigma, \mathfrak{l}_{\mathfrak{r}}, r) \;\middle|\; \begin{array}{l} up(\sigma) = \mathtt{Some}(\sigma') \\ \wedge\, \sigma', \mathfrak{u}_{2,\mathfrak{r}} \Downarrow_0 r_1 \\ \wedge\, next(\sigma, r_1) = \mathtt{Some}(\sigma'') \\ \wedge\, \sigma'', \mathfrak{n}_{2,\mathfrak{r}} \Downarrow_0 \mathtt{Some}(r) \end{array}\right\}
\end{array}
\tag{4.1}
$$

The final evaluation relation is then computed step by step using the immediate consequence $\mathcal{F}$. It extends the evaluation relation $\Downarrow_0$ to the new relation $\mathcal{F}(\Downarrow_0)$ below. A semantic triple is accepted if at least one rule $\mathfrak{r}$ generates it through $apply_{\mathfrak{r}}(\Downarrow_0)$.

$$\mathcal{F}(\Downarrow_0) = \{(\sigma, t, r) \mid \exists \mathfrak{r}, cond_{\mathfrak{r}}(\sigma) \wedge (\sigma, t, r) \in apply_{\mathfrak{r}}(\Downarrow_0)\}$$

Each application of $\mathcal{F}$ extends the relation $\Downarrow_0$ with an extra step in derivations: $\mathcal{F}(\varnothing)$ is the set of all semantic triples (σ, t, r) generated by axioms, and $\mathcal{F}^n(\varnothing)$ is the set of all semantic triples (σ, t, r) built by derivations whose depth is less or equal than n.

We can equip the set of evaluation relations $\mathcal{P}(st \times term \times res)$ with the set inclusion lattice structure (see Section 3.2.2). In this lattice, the functions $apply_{\mathfrak{r}}$ and $\mathcal{F}$ are monotonic: we can thus consider the fixed points of $\mathcal{F}$. There are several interesting fixed points. The least fixed point $\Downarrow_{lfp}$ contains the semantic triples which can be derived from a finite derivation; the greatest fixed point $\Downarrow_{gfp}$ contains the semantic triples which can be derived from

```
Inductive eval : st → term → res → Type :=
  | eval_cons : forall sigma t r n,
      t = left n →
      cond n sigma →
      apply n sigma r →
      eval sigma t r
with apply : name → st → res → Type :=
  | apply_Ax : forall n ax sigma r,
      rule_struct n = Rule_struct_Ax _ →
      rule n = Rule_Ax ax →
      ax sigma = Some r →
      apply n sigma r
  | apply_R1 : forall n t up sigma sigma' r,
      rule_struct n = Rule_struct_R1 t →
      rule n = Rule_R1 _ up →
      up sigma = Some sigma' →
      eval t sigma' r →
      apply n sigma r
  | apply_R2 : forall n t1 t2 up next
                 sigma sigma1 sigma2 r r',
      rule_struct n = Rule_struct_R2 t1 t2 →
      rule n = Rule_R2 up next →
      up sigma = Some sigma1 →
      eval t1 sigma1 r →
      next sigma r = Some sigma2 →
      eval t2 sigma2 r' →
      apply n sigma r'.
```

Program 4.2: Coq definition of the concrete semantics $\Downarrow$

finite and infinite derivations. As said above, we are not interested in non-terminating behaviours of programs in this dissertation: the concrete semantics is defined as the least fixed point $\Downarrow_{lfp}$, which corresponds to an inductive interpretation of the rules. We write it $\Downarrow$. No semantics is given to non-terminating programs.

The Coq implementation of $\Downarrow$ is shown in Program 4.2. It directly builds the fixed point as an inductive definition. The Coq function `rule` fetches the transfer functions whilst `rule_struct` fetches the syntactic aspects of the semantics, such as the different terms $\mathfrak{u}_1$, $\mathfrak{u}_2$, and $\mathfrak{n}_2$. The predicates `eval` and `apply` are defined as mutually recursive inductive definitions, indicated by the **`with`** construct.

4.4 Abstract Semantics

The purpose of mechanising the pretty-big-step semantics is to facilitate the soundness proof of static analysers with respect to a concrete semantics. We thus provide a mechanised way to define an abstract semantics and prove it sound with respect to the concrete one. Its usage to build and prove static analysers is described in Section 4.6.

As stated in Section 3.3, the starting point for our development is the abstract interpretation of big-step semantics laid out by Schmidt [Sch95]. In this section, we describe how an adapted version of Schmidt's framework can be implemented using the Coq proof assistant. There are several steps in such a formalisation:

- define the connection relating concrete and abstract domains of semantic contexts and results. Abstract interpretation is usually based on Galois connections, but as explained in Section 3.2.4, we have lightened this connection to decidable posets related by a concretisation function.
- based on the connection between concrete and abstract domains, prove the local soundness: the side-conditions and transfer functions of each concrete rule are soundly abstracted by their abstract counterpart.
- given the local soundness, prove the global soundness: the abstract semantics $\Downarrow^\sharp$ is a sound approximation of the concrete semantics $\Downarrow$.

The concretisation functions γ relate the concrete and abstract semantic triples (σ, t, r) and $\left(\sigma^\sharp, t, r^\sharp\right)$. They let us state and prove the property relating the concrete and the abstract semantics: let $t \in term$, $\sigma \in st$, $\sigma^\sharp \in st^\sharp$, $r \in res$ and $r^\sharp \in res^\sharp$,

$$\text{if} \begin{cases} \sigma \in \gamma\left(\sigma^\sharp\right) \\ \sigma, t \Downarrow r \\ \sigma^\sharp, t \Downarrow^\sharp r^\sharp \end{cases} \quad \text{then } r \in \gamma\left(r^\sharp\right).$$

We illustrate the development using the language and domains presented in Section 4.1. However, we emphasize that the approach is generic: once an abstract domain is given, and abstract transfer functions are shown to be sound, then the full abstract semantics is sound by construction. We do not detail how to define and prove sound abstract transfer functions, as it is usually not perceived as a difficult task once the domains have been defined. In the case of a large semantics such as JSCert, however, any automation technique which could help this task would be highly valuable. We do believe that such an automation technique can be adapted to our framework, given for instance the work of Van Horn and Might [VM11a] and of Midtgaard and Jensen [MJ08] on this subject.

4.4.1 Rule Abstraction

The game-changing aspect of this formalisation is that each abstract rule is directly built from its corresponding concrete rule. Instead of building ad-hoc abstract rules such as the ones shown in Section 4.2, each concrete rule is abstracted separately, replacing the semantic domains by their abstract counterparts but leaving the structure unchanged. This formalisation choice not only guides the definition of the abstract semantics, it also makes the abstract and concrete semantics much closer to each other. This stronger correspondance contributes in making the soundness proof generic.

$+^\sharp$	$\bot$	$-$	0	$+$	$-_0$	$\pm$	$+_0$	$\top_{\mathbb{Z}}$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$\bot$	$-$	$-$	$\top_{\mathbb{Z}}$	$-$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$
0	$\bot$	$-$	0	$+$	$-_0$	$\pm$	$+_0$	$\top_{\mathbb{Z}}$
$+$	$\bot$	$\top_{\mathbb{Z}}$	$+$	$+$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$+$	$\top_{\mathbb{Z}}$
$-_0$	$\bot$	$-$	$-_0$	$\top_{\mathbb{Z}}$	$-_0$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$
$\pm$	$\bot$	$\top_{\mathbb{Z}}$	$\pm$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$
$+_0$	$\bot$	$\top_{\mathbb{Z}}$	$+_0$	$+$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$+_0$	$\top_{\mathbb{Z}}$
$\top_{\mathbb{Z}}$	$\bot$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$	$\top_{\mathbb{Z}}$

Figure 4.6: Table of the $+^\sharp$ abstract operation on the *Sign* domain

4.4.1.1 Abstract Rules

Here follows the concrete and abstract versions of Rule $\textsc{red-asn}(\mathsf{x}, e)$. Note that although the second rule is abstract, we still note $\Downarrow$ the derivation relation in the rule. This is due to the notation of rules as data structures: the symbol $\Downarrow$ present in the rules is meant to be seen as a notational convention and to be differentiated from the predicates $\Downarrow$ and $\Downarrow^\sharp$.

$$\textsc{red-asn}(\mathsf{x}, e)\quad \frac{E, e \Downarrow r \qquad E, r, \mathsf{x} :=_1 \cdot \Downarrow r'}{E, \mathsf{x} := e \Downarrow r'} \qquad\qquad \textsc{red-asn}(\mathsf{x}, e)\quad \frac{E^\sharp, e \Downarrow r^\sharp \qquad E^\sharp, r^\sharp, \mathsf{x} :=_1 \cdot \Downarrow r'^\sharp}{E^\sharp, \mathsf{x} := e \Downarrow r'^\sharp}$$

The concrete and abstract rules $\textsc{red-asn}(\mathsf{x}, e)$ are very similar with each other, thanks to the local abstraction. This way of locally abstracting rules is novel. It seems to correspond to the minimal amount of effort needed to abstract a semantic in general: we only request to abstract the side-conditions and the transfer functions, that is, the operations on the domains. In particular, the rule names, the rule structure, and the syntactic terms are left unchanged. The Coq development has been designed so that the concrete and the abstract rules share the most definitions possible. This is why the types *st* and *res* are parts of the `semantics` Coq definition (see Section 4.3.1): both the concrete and abstract version are based on the same `structure` record, but they come with their own `semantics` record.

For the specific rule $\textsc{red-asn}(\mathsf{x}, e)$, there is not really any operation to be abstracted: the *cond* predicate is trivial, as well as the transfer functions. Let us thus consider instead Rule $\textsc{red-asn-1}(\mathsf{x})$. This rule receives in the concrete world a couple of an environment and a result $r \in Out_e$ (computed from the expression e), and only applies if this result is a value. In the abstract world, however, an abstract value can represent both a value and an error (consider for instance the abstract result $(\pm, err^\sharp)$). This is solved by defining an abstract rule accepting any result, but filtering the value part of this result. The local soundness of this abstract rule is proven later: what is important to note here is that the

```
Definition left_term n : terms :=
  match n with
  | name_add e1 e2 => expr_add e1 e2
  | name_add_1 e2 => expr_add_1 e2
  | name_add_2 => expr_add_2
  | name_if e s1 s2 => stat_if e s1 s2
  | name_if_1_pos s1 s2 => stat_if_1 s1 s2
  | name_if_1_neg s1 s2 => stat_if_1 s1 s2
  (* ... *)
  end.
```

(a) The Coq implementation of the $\mathfrak{l}$ function

```
Definition struct_rule n : Rule_struct terms :=
  match n with
  | name_add e1 e2 => Rule_struct_R2 (term := terms) e1 (expr_add_1 e2)
  | name_add_1 e2 => Rule_struct_R2 (term := terms) e2 expr_add_2
  | name_add_2 => Rule_struct_Ax _
  | name_if e s1 s2 => Rule_struct_R2 (term := terms) e (stat_if_1 s1 s2)
  | name_if_1_pos s1 s2 => Rule_struct_R1 (term := terms) s1
  | name_if_1_neg s1 s2 => Rule_struct_R1 (term := terms) s2
  (* ... *)
  end.
```

(b) The Coq implementation of the terms $\mathfrak{u}_1$, $\mathfrak{u}_2$, and $\mathfrak{n}_2$

Program 4.3: Snippet of the syntactic components

abstraction of Rule RED-ASN-1(x) is defined and proven sound independently of the whole semantics (which is why we qualify the local soundness as local).

$$\text{RED-ASN-1}(\mathsf{x})\quad \frac{}{E, v, \mathsf{x} :=_1 \cdot \Downarrow E\left[\mathsf{x} \leftarrow v\right]} \qquad\qquad \text{RED-ASN-1}(\mathsf{x})\quad \frac{}{E^\sharp, r^\sharp, \mathsf{x} :=_1 \cdot \Downarrow E^\sharp\left[\mathsf{x} \leftarrow r^\sharp|_{Val^\sharp}\right]}$$

Some abstract rules need to abstract concrete operations. For instance, Rule RED-ADD-2 below is abstracted by a rule using an abstract operator $+^\sharp$ instead of the concrete addition of integers. The table of the $+^\sharp$ operator is given in Figure 4.6. As for Rule RED-ASN-1(x), abstract Rule RED-ADD-2 considers a general result $r_2^\sharp$ and filter its value part.

$$\text{RED-ADD-2}\quad \frac{}{E, v_1, v_2, \cdot +_2 \cdot \Downarrow v_1 + v_2} \qquad\qquad \text{RED-ADD-2}\quad \frac{}{E^\sharp, v_1^\sharp, r_2^\sharp, \cdot +_2 \cdot \Downarrow v_1^\sharp +^\sharp r_2^\sharp|_{Val^\sharp}}$$

We now present how these rules are implemented in Coq. Program 4.3 shows the syntactic component of various rules of Figure 3.4. It is shared between the concrete and the abstract semantics. Program 4.4 shows the definition of the concrete transfer functions of the rules for the addition. Some monads have been inlined, but this does not change the message.

```
Definition concrete_rule n : Rule st res :=
  match n with
  | name_add e1 e2 =>
    let up :=
      if_st_expr (fun E =>
        Some (st_expr E)) in
    let next sigma r :=
      if_st_expr (fun E =>
        Some (st_add_1 E r)) sigma in
    Rule_R2 up next
  | name_add_1 e2 =>
    let up sigma :=
      match sigma with
      | st_add_1 E r => Some (st_expr E)
      | _ => None
      end in
    let next sigma r :=
      match sigma with
      | st_add_1 E (oute_val v) => Some (st_add_2 v r)
      | _ => None
      end in
    Rule_R2 up next
  | name_add_2 =>
    let ax sigma :=
      match sigma with
      | st_add_2 v1 (oute_val v2) => Some (oute_val (v1 + v2) : res)
      | _ => None
      end in
    Rule_Ax ax
  (* ... *)
  end.
```

Program 4.4: Snippet of the concrete *rule* function

```
Definition acond n asigma : Prop :=
  match n, asigma with
  | _, sign_ast_top =>
    True
  | name_if e s1 s2, ast_stat aE =>
    True
  | name_if_1_pos s1 s2, ast_if_1 aE ar =>
    ares_expr (Sign.pos, ⊥♯) ⊑ ar
  | name_if_1_neg s1 s2, ast_if_1 aE ar =>
    ares_expr (Sign.zero, ⊥♯) ⊑ ar ∨ ares_expr (Sign.neg, ⊥♯) ⊑ ar
  (* ... *)
  end.
```

Program 4.5: Snippet of the $cond^\sharp$ predicate

Program 4.5 is a snippet from the CoQ formalisation showing some abstract side-conditions. They correspond in a one-to-one fashion to the rules of the concrete semantics defining the *cond* predicate. For instance, Line 8 states that the abstract Rule RED-IF-1-POS(s_1, s_2) applies when $+ \sqsubseteq r^\sharp$, where $r^\sharp$ is the result computed from the *if* condition. This is a sound definition as for all $v \in \gamma\left(v^\sharp\right)$ such that $v > 0$, we have $+ \sqsubseteq v^\sharp$: the abstract Rule RED-IF-1-POS(s_1, s_2) applies when the concrete Rule RED-IF-1-POS(s_1, s_2) does. The coercion on $+$ has been removed in the CoQ snippet: the symbol $\bot\sharp$ of Line 8 represents the error component $\overline{err}^\sharp$ of abstract expression outputs.

The CoQ snippet of Program 4.6 shows the encoding of the abstract rules for the addition—Rules RED-ADD(e_1, e_2), RED-ADD-1(e_2), and RED-ADD-2. Note how Line 21 filters out the error component, as in the abstract rule RED-ADD-2 above. As for JSREF (see Section 2.6.2), monads are used to check whether the semantic context is in the expected form, extracting the relevant information. Some of these monads are shown in Program 4.7. Because the concrete and the abstract semantics share a lot, the definition of the abstract semantics is guided by the concrete semantics. For instance, note how close Programs 4.4 and 4.6 are. This considerably helps both the definition of rules and their soundness proofs.

Abstracting the rules of a semantics by only changing their semantic parts, leaving their syntactic part unchanged, is appealing. But it does not seem sound as-is. For instance, abstract Rule RED-ASN-1$(\mathtt{x})$ above only considers non-error results and seems to suffer from the same issue than Rule IF-ABS of Section 3.6. As explained in Section 3.3, the soundness of an abstract semantics in intimately linked to how it covers the concrete semantics. There are several ways in which coverage can be ensured. One way is to add a number of ad-hoc rules, such as the rules seen in Section 4.2. Instead, we follow here an approach where we obtain coverage in a systematic fashion, by changing how abstract rules are combined to form the abstract semantics. This is described in Section 4.4.2 below.

4.4.1.2 Local Soundness

The advantage of our method does not limit to guiding the definition of abstract rules, but also guiding their proof. In particular, abstract rules are requested to be related to the concrete rule of the same name, without considering any other rule. Let us now see what are the conditions we impose on abstract rules. As the only changing part between the abstract and the concrete rules consists of the semantic parts, there are two conditions: one for side-conditions, and one for transfer functions.

Side-conditions describe when a given rule applies. As we have seen in Section 3.3, the soundness of the abstract semantics is provided by the coverage of the abstract semantics with respect to the concrete semantics. The requested property about side-conditions is thus that whenever there is a state in the concretisation of an abstract state $\sigma^\sharp$ which

```
Definition arule n : Rule ast ares :=
  match n with
  | name_add e1 e2 =>
    let up :=
      if_ast_expr (fun E =>
        Some (ast_expr E)) in
    let next asigma o :=
      if_ast_expr (fun E =>
        Some (ast_add_1 E o)) asigma in
    Rule_R2 up next
  | name_add_1 e2 =>
    let up :=
      if_ast_add_1 (fun E av =>
        Some (ast_expr E)) in
    let next asigma1 ar2 :=
      if_ast_add_1 (fun E (av1, err) =>
        Some (ast_add_2 av1 ar2)) asigma in
    Rule_R2 up next
  | name_add_2 =>
    let ax :=
      if_ast_add_2 (fun av1 (av2, err) =>
        Some (ares_expr (Sign.sem_add av1 av2, ⊥♯))) in
    Rule_Ax ax
  (* ... *)
  end.
```

Program 4.6: Snippet of the abstract *rule* function

```
Definition if_ast_expr A (f : aEnv → option A) (asigma : ast) :=
  match asigma return option A with
  | ast_expr E => f E
  | ast_top => f (⊤♯)
  | _ => None
  end.

Definition if_ast_add_1 A (f : aEnv → aVal → option A) (asigma : ast) :=
  match asigma return option A with
  | ast_add_1 E ar => if_ares_expr (f E) ar
  | ast_top => f (⊤♯) (⊤♯)
  | _ => None
  end.

Definition if_ast_add_2 A (f : aVal → aVal → option A) (asigma : ast) :=
  match asigma return option A with
  | ast_add_2 av1 ar => if_ares_expr (f av1) ar
  | ast_top => f (⊤♯) (⊤♯)
  | _ => None
  end.
```

Program 4.7: Definition of the monads used in Program 4.6

```
Inductive propagates : (ast → Prop) → (st → Prop) → aRule → Rule → Prop :=
  | propagates_Ax : forall cond acond ax aax,
      (forall sigma asigma r ar,
        cond sigma → acond asigma →
        gst asigma sigma →
        ax sigma = Some r →
        aax asigma = Some ar →
        gres ar r) →
      propagates acond cond (Rule_Ax aax) (Rule_Ax ax)
  | propagates_R1 : forall cond acond up aup,
      (forall sigma asigma sigma' asigma',
        cond sigma → acond asigma →
        gst asigma sigma →
        up sigma = Some sigma' →
        aup asigma = Some asigma' →
        gst asigma' sigma') →
      propagates acond cond (Rule_R1 _ aup) (Rule_R1 _ up)
  | propagates_R2 : forall cond acond up aup next anext,
      (forall sigma asigma sigma' asigma',
        cond sigma → acond asigma →
        gst asigma sigma →
        up sigma = Some sigma' →
        aup asigma = Some asigma' →
        gst asigma' sigma') →
      (forall sigma asigma r ar sigma' asigma',
        cond sigma → acond asigma →
        gst asigma sigma → gres ar r →
        next sigma r = Some sigma' →
        anext asigma ar = Some asigma' →
        gst asigma' sigma') →
      propagates acond cond (Rule_R2 aup anext) (Rule_R2 up next).
```

Program 4.8: Propagation of abstraction through transfer functions

would trigger a concrete rule, the corresponding abstract rule is also triggered by $\sigma^\sharp$. The soundness criterion for the side-condition of Rule $\mathfrak{r}$ follows.

$$\forall \sigma, \sigma^\sharp.\, \sigma \in \gamma(\sigma^\sharp) \to cond_{\mathfrak{r}}(\sigma) \to cond^\sharp_{\mathfrak{r}}(\sigma^\sharp) \tag{4.2}$$

This criterion is expressed in Coq as follows. The predicate `gst` is the concretisation function (expressed as a predicate) for semantic contexts.

```
Hypothesis acond_sound : forall n asigma sigma,
  gst asigma sigma → cond n sigma → acond n asigma.
```

The criteria for transfer functions state that the abstraction is propagated along transfer functions—in particular, no concrete state can be missed by moving along transfer functions: this is exactly what the coverage notion of Schmidt imposes in this setting. These criteria are defined as a relation ~ between rules (called `propagates` in the Coq files), made precise below. For example, concrete and abstract axioms ax and $ax^\sharp$ are both functions from semantic contexts to results, one in the concrete domain and the other in the abstract domain. They intuitively satisfy the following criterion.

$$\forall \sigma, \sigma^\sharp.\ \sigma \in \gamma\left(\sigma^\sharp\right) \to ax\left(\sigma\right) \in \gamma\left(ax^\sharp\left(\sigma^\sharp\right)\right) \tag{4.3}$$

Criterion 4.3 is very strong. In the soundness proof of Section 4.4.3, we only apply this criterion when the side-condition applies and when both concrete and abstract transfer functions are defined. We thus lighten this constraint to Criterion 4.4 below. Program 4.8 shows the definition of the `propagates` predicate which enforces the criteria 4.4, 4.5, and 4.6 over transfer functions. The predicates `gst` and `gres` are the concretisation functions for semantic contexts and results. Note that Line 4 requires that both $cond$ and $cond^\sharp$ accept the rule, but $cond^\sharp$ follows by criterion 4.2. It is nonetheless added for clarity.

The relation ~ relates concrete and abstract rules. It is defined as follows.

- A concrete and an abstract axioms $ax : st \to res$ and $ax^\sharp : st^\sharp \to res^\sharp$ are related if and only if for all σ and $\sigma^\sharp$ on which both functions ax and $ax^\sharp$ are defined, such that the concrete rule applies on σ, and such that $\sigma \in \gamma\left(\sigma^\sharp\right)$, then $ax\left(\sigma\right) \in \gamma\left(ax^\sharp\left(\sigma^\sharp\right)\right)$.

$$\begin{aligned}&\forall \sigma^\sharp, \sigma \in \gamma\left(\sigma^\sharp\right).\ cond_{\mathfrak{r}}\left(\sigma\right) \to \\ &\qquad ax\left(\sigma\right) \text{ and } ax^\sharp\left(\sigma^\sharp\right) \text{ defined} \to ax\left(\sigma\right) \in \gamma\left(ax^\sharp\left(\sigma^\sharp\right)\right)\end{aligned} \tag{4.4}$$

- A concrete and an abstract format 1 rules $up : st \to st$ and $up^\sharp : st^\sharp \to st^\sharp$ are related if and only if for all σ and $\sigma^\sharp$ on which both functions up and $up^\sharp$ are defined, if the concrete rule applies on σ and $\sigma \in \gamma\left(\sigma^\sharp\right)$, then $up\left(\sigma\right) \in \gamma\left(up^\sharp\left(\sigma\right)\right)$.

$$\begin{aligned}&\forall \sigma^\sharp, \sigma \in \gamma\left(\sigma^\sharp\right).\ cond_{\mathfrak{r}}\left(\sigma\right) \to \\ &\qquad up\left(\sigma\right) \text{ and } up^\sharp\left(\sigma^\sharp\right) \text{ defined} \to up\left(\sigma\right) \in \gamma\left(up^\sharp\left(\sigma^\sharp\right)\right)\end{aligned} \tag{4.5}$$

- For format 2 rules, we impose the same condition on the up and $up^\sharp$ transfer functions than above, and we add the additional condition over the transfer functions $next : st \to res \to st$ and $next^\sharp : st^\sharp \to res^\sharp \to st^\sharp$: for all σ, $\sigma^\sharp$, r, and $r^\sharp$ on which both functions $next$ and $next^\sharp$ are defined, such that the concrete rule applies on σ, and such that $\sigma \in \gamma\left(\sigma^\sharp\right)$ and $r \in \gamma\left(r^\sharp\right)$, then $next\left(\sigma, r\right) \in \gamma\left(next^\sharp\left(\sigma^\sharp, r^\sharp\right)\right)$.

$$\begin{aligned}&\forall \sigma^\sharp, \sigma \in \gamma\left(\sigma^\sharp\right), r^\sharp, r \in \gamma\left(r^\sharp\right).\ cond_{\mathfrak{r}}\left(\sigma\right) \to \\ &\quad next\left(\sigma, r\right) \text{ and } next^\sharp\left(\sigma^\sharp, r^\sharp\right) \text{ defined} \to next\left(\sigma, r\right) \in \gamma\left(next^\sharp\left(\sigma^\sharp\right), r^\sharp\right)\end{aligned} \tag{4.6}$$

Let us now consider how these criteria manifest in the context of Rule RED-ASN-1(x) of Section 4.4.1.1. The oddity of this rule is that the given result $r^\sharp$ can represent both an error and a value, as in $\left(\pm, err^\sharp\right)$. We can see that the value $42 \in \gamma\left(\left(\pm, err^\sharp\right)\right)$ triggers the side-condition $cond_{\text{RED-ASN-1(x)}}\left(42\right)$: by criteria 4.2, the abstract side-condition is thus forced

```
Lemma if_st_expr_out : forall A (f : env → option A) sigma r,
  if_st_expr f sigma = Some r →
  exists E, sigma = st_expr E ∧ f E = Some r.

Lemma if_ast_expr_out : forall A (f : aEnv → option A) sigma r,
  if_ast_expr f sigma = Some r →
  (sigma = ast_top ∧ f (⊤♯) = Some r) ∨
  exists E, sigma = ast_expr E ∧ f E = Some r.
```

Program 4.9: Lemmata about monadic constructors

to apply to $\left(\pm, err^\sharp\right)$. The second surprising effect of the abstract rule is that it filters the given result $r^\sharp$ to only get the non-error ones $r^\sharp|_{Val^\sharp}$. This would break Criterion 4.3, as $err \in \gamma\left(\left(\pm, err^\sharp\right)\right)$. But Criterion 4.3 has been refined to Criterion 4.4, which requires the semantic contexts σ to trigger the rule. As err does not trigger Rule RED-ASN-1(x), filtering out errors is not a problem for Criterion 4.4.

The structure of the soundness proof of abstract rules with respect to concrete rules is similar to the correctness proof of JSRef with respect to JSCert. Each monadic construct is associated with a behaviour lemma, as for JSRef in Program 2.16. For instance, Program 4.9 shows the behaviour lemmata of the concrete `if_st_expr` and abstract `if_ast_expr` monads. There is one significant difference with JSCert: the concrete and the abstract rules share the same data structure. This considerably eases the proof of the abstract rules. Proving the local soundness of rules is a relatively easy task (although it highly depends on the chosen domain). We now show how we can build a semantics from such abstract rules.

4.4.2 Inference Trees

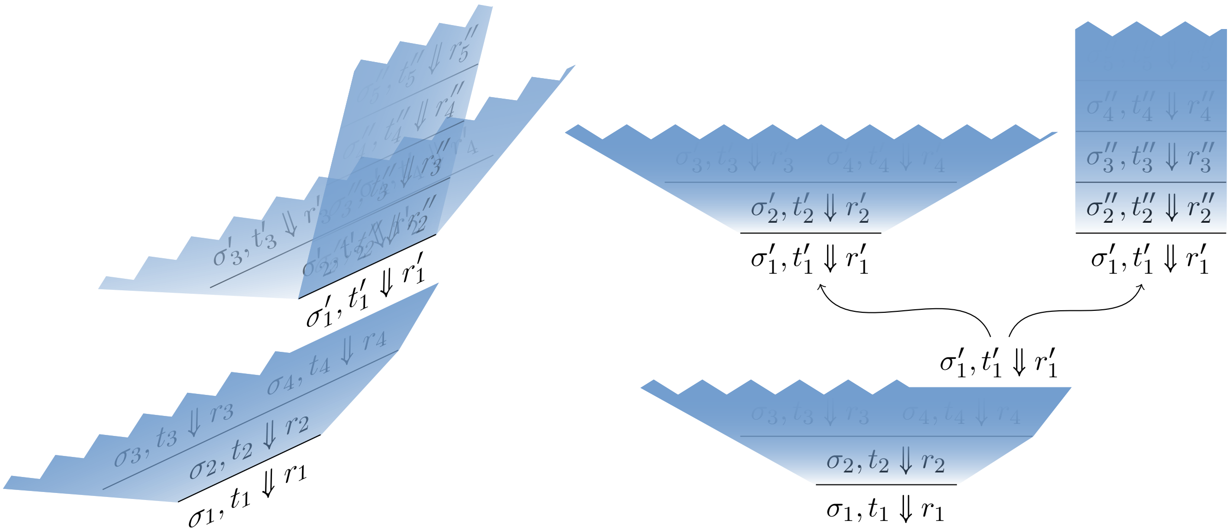
Concrete and abstract inference rules share by design the same structure. However, the semantics given to a set of abstract rules differs from the concrete semantics defined in Section 4.3.2. This difference manifests itself in the way that rules are assembled. In a nutshell, the abstract semantics $\Downarrow^\sharp$ applies every applicable rules instead of just one.

4.4.2.1 Abstract Immediate Consequence

As in Section 4.3.2, we define an operator $\mathcal{F}^\sharp$, the abstract immediate consequence, which infers new derivations from a set of already established derivations.

$$\mathcal{F}^\sharp : \mathcal{P}\left(st^\sharp \times term \times res^\sharp\right) \to \mathcal{P}\left(st^\sharp \times term \times res^\sharp\right)$$

The *apply* function can still be used to apply a rule: it has been defined independently of the domain and can manipulate abstract values. It now implicitly uses the abstract transfer functions instead of the concrete ones. The definition of the function $\mathcal{F}^\sharp$ differs in one



(a) Three dimensional representation

(b) Flat representation

Figure 4.7: Intuition behind abstract derivations

important aspect from its concrete counterpart: in order to obtain coverage of concrete rules, $\mathcal{F}^\sharp$ must apply *all* the applicable rules.

$$\mathcal{F}^\sharp\left(\Downarrow^\sharp_0\right) = \left\{ \left(\sigma^\sharp, t, r^\sharp\right) \;\middle|\; \begin{array}{l} \forall \mathfrak{r}.\, t = \mathfrak{l}_\mathfrak{r} \to cond^\sharp_\mathfrak{r}(\sigma) \to \\ \quad \left(\sigma^\sharp, t, r^\sharp\right) \in apply_\mathfrak{r}\left(\Downarrow^\sharp_0\right) \end{array} \right\}$$

In other words, the operator $\mathcal{F}^\sharp$ extends its relation argument $\Downarrow^\sharp_0$ by adding the triples $\left(\sigma^\sharp, t, r^\sharp\right)$ such that the result $r^\sharp$ is valid for *all* applicable rules. By defining $\mathcal{F}^\sharp$ in this way, we avoid having to add ad-hoc rules such as Rule IF-ABS-CORRECTED from Section 4.2: a correct result is one which includes the computation from both branches. One way of picturing this is by imagining a three dimensional derivation as in Figure 4.7a: the construction of the derivation forks into two separate branches, and each branch has to accept the current triple for it to be correct. Representing derivations in three dimensions hinders readability: we shall use the arrow notation of Figure 4.7b to represent such forks.

We now consider an example. The program $if\ (\mathsf{x} > 0)\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18)$ always sets r to a positive value if x is defined, but our framework only partially gets this result. Let us analyse it in an environment $E^\sharp_1 \in Env^\sharp$ where x is $+ \in Sign$, and in an environment $E^\sharp_2 \in Env^\sharp$ where x is $\top_\mathbb{Z}$, that is, x is defined but we know nothing about its value. Both derivations are shown in Figure 4.8. In either case, it expands to $if_1\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18)$, and carries an information about the computed expression x which is either $+$ or $\top_\mathbb{Z}$. In the first case we know that this expression is positive, and only Rule RED-IF-1-POS$(\mathsf{r} := \mathsf{x}, \mathsf{r} := 18)$ applies: we evaluate $\mathsf{r} := \mathsf{x}$ to the environment $E^{\sharp\prime}_1$ in which r is positive. However in the second case, we do not know which branch will be executed and thus execute both: we also apply Rule RED-IF-1-NEG$(\mathsf{r} := \mathsf{x}, \mathsf{r} := 18)$, which executes $\mathsf{r} := 18$ and sets r to $+$. In

$$
\textsc{red-if}(\mathsf{x}, \mathsf{r} := \mathsf{x}, \mathsf{r} := 18)\;
\dfrac{
\textsc{red-var}(\mathsf{x})\,\dfrac{}{E_1^\sharp, \mathsf{x} \Downarrow +}
\qquad
\textsc{red-if-1-pos}(\mathsf{r} := \mathsf{x}, \mathsf{r} := 18)\;
\dfrac{
\textsc{red-asn}(\mathsf{r}, \mathsf{x})\;
\dfrac{
\textsc{red-var}(\mathsf{x})\,\dfrac{}{E_1^\sharp, \mathsf{x} \Downarrow +}
\qquad
\textsc{red-asn-1}(\mathsf{r})\,\dfrac{}{E_1^\sharp, +, \mathsf{r} :=_1 \cdot \Downarrow {E_1^\sharp}'}
}{E_1^\sharp, \mathsf{r} := \mathsf{x} \Downarrow {E_1^\sharp}'}
}{E_1^\sharp, +, \mathit{if}_1\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18) \Downarrow {E_1^\sharp}'}
}{E_1^\sharp, \mathit{if}\ (\mathsf{x} > 0)\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18) \Downarrow {E_1^\sharp}'}
$$

(a) In an abstract environment $E_1^\sharp$ where x is $+$

$$
\textsc{red-if-1-neg}(\mathsf{r} := \mathsf{x}, \mathsf{r} := 18)\;
\dfrac{
\textsc{red-asn}(\mathsf{r}, 18)\;
\dfrac{
\textsc{red-const}(18)\,\dfrac{}{E_2^\sharp, 18 \Downarrow +}
\qquad
\textsc{glue-weaken}\;
\dfrac{
\textsc{red-asn-1}(\mathsf{r})\,\dfrac{}{E_2^\sharp, \top_{\mathbb{Z}}, \mathsf{r} :=_1 \cdot \Downarrow {E_2^\sharp}'}
}{E_2^\sharp, +, \mathsf{r} :=_1 \cdot \Downarrow {E_2^\sharp}'}
}{E_2^\sharp, \mathsf{r} := 18 \Downarrow {E_2^\sharp}'}
}{E_2^\sharp, \top_{\mathbb{Z}}, \mathit{if}_1\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18) \Downarrow {E_2^\sharp}'}
$$

$$
\textsc{red-if-1-pos}(\mathsf{r} := \mathsf{x}, \mathsf{r} := 18)\;
\dfrac{
\textsc{red-asn}(\mathsf{r}, \mathsf{x})\;
\dfrac{
\textsc{red-var}(\mathsf{x})\,\dfrac{}{E_2^\sharp, \mathsf{x} \Downarrow \top_{\mathbb{Z}}}
\qquad
\textsc{red-asn-1}(\mathsf{r})\,\dfrac{}{E_2^\sharp, \top_{\mathbb{Z}}, \mathsf{r} :=_1 \cdot \Downarrow {E_2^\sharp}'}
}{E_2^\sharp, \mathsf{r} := \mathsf{x} \Downarrow {E_2^\sharp}'}
}{E_2^\sharp, \top_{\mathbb{Z}}, \mathit{if}_1\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18) \Downarrow {E_2^\sharp}'}
$$

$$
\textsc{red-if}(\mathsf{x}, \mathsf{r} := \mathsf{x}, \mathsf{r} := 18)\;
\dfrac{
\textsc{red-var}(\mathsf{x})\,\dfrac{}{E_2^\sharp, \mathsf{x} \Downarrow \top_{\mathbb{Z}}}
\qquad
E_2^\sharp, \top_{\mathbb{Z}}, \mathit{if}_1\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18) \Downarrow {E_2^\sharp}'
}{E_2^\sharp, \mathit{if}\ (\mathsf{x} > 0)\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18) \Downarrow {E_2^\sharp}'}
$$

(b) In an abstract environment $E_2^\sharp$ where x is $\top_{\mathbb{Z}}$

Figure 4.8: Two derivations starting from the program $\mathit{if}\ (\mathsf{x} > 0)\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18)$

the first branch, however, we execute $\mathsf{r} := \mathsf{x}$ in $E_2^\sharp$ and sets r to $\top_{\mathbb{Z}}$ to get ${E_2^\sharp}'$. The result is different for both branches, and we need a construct to build this derivation: Figure 4.8b uses Rule GLUE-WEAKEN of Section 5.1.1 in the branch of Rule RED-IF-1-NEG$(\mathsf{r} := \mathsf{x}, \mathsf{r} := 18)$ to complete this gap.

Although Figure 4.8b is not precise enough to ensure that the variable r is positive at the end of the execution, it is precise enough to ensure that no error is thrown: we can prove that Rule RED-ERROR-STAT$(\mathit{if}_1\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18))$ never triggers. This example illustrates a shortcoming of our approach: even though we know that the tested value has to be positive in the positive branch, there is no information about how this value was computed (evaluating x in this example). The non-local information which would have allowed to deduce that x is positive in the environment is not available in this framework as-is. This will be fixed by rules such as Rule GLUE-TRACE-PARTITIONING of Chapter 5.

The goal of this example was to show the locality of the approach: it provides a simple way of building large abstract semantics sound by construction, but some work is left to make it catch some non-local behaviours. Chapter 5 aims at catching some of these non-local behaviours. For instance, Figure 5.3 is a precise version of Figure 4.8b. This formalisation is however already able to provide the equivalent of rules such as Rule WHILE-ABS-FIXED-POINT of Section 4.2, as we now explore.

4.4.2.2 Finite and Infinite Derivation Trees

The function $\mathcal{F}^\sharp$ is a monotone function on the powerset lattice $\mathcal{P}\left(st^\sharp \times term \times res^\sharp\right)$. The least fixed point of $\mathcal{F}^\sharp$ (with respect to the inclusion order $\subseteq$) corresponds to all triples which can be inferred using finite abstract derivation trees. These triples represent valid properties of the program, but the restriction to finite derivations means that certain properties can not be inferred. In particular, a finite abstract derivation is a proof that the program always terminates. Requiring such a proof can be very limiting.

Consider the program $while\ (\mathsf{x} > 0)\ (\mathsf{x} := \mathsf{x} + (-1))$ evaluated on an abstract context where x is $+$. Its concrete derivation clearly terminates, but there is no finite abstract derivation in the sign abstraction to witness it. Indeed, initially x is bound to $+$; after the first iteration it is bound to $+_0$; after the second iteration, it is bound to $\top_\mathbb{Z}$ (for the same reason that the derivation of Figure 4.8b); then its value becomes stable at $\top_\mathbb{Z}$. Every subsequent iteration has thus to consider the case where x is $\top_\mathbb{Z}$—and in particular strictly positive—and has to compute an additional iteration. Hence, there is no finite abstract derivation: the abstract domain is not precise enough. Another way to understand this result is that an abstract derivation tree has to cover every concrete derivation tree: since there is no bound on the number of execution steps of the concrete derivation (which depends on the initial value of x, the loop being unfolded this many times), abstract derivations have to be infinite.

Figure 4.9 depicts the abstract derivation tree built by recursively applying $\mathcal{F}^\sharp$ from an environment in which x is $\top_\mathbb{Z}$: we know that x is defined, but we know nothing about its value. Both rules RED-WHILE-1-POS$(\mathsf{x}, \mathsf{x} := \mathsf{x} + (-1))$ and RED-WHILE-1-NEG$(\mathsf{x}, \mathsf{x} := \mathsf{x} + (-1))$ are executed, and their result $\{\mathsf{x} \mapsto \top_\mathbb{Z}\}$ is propagated. This follows the definition of $\mathcal{F}^\sharp$, which applies *every* rule which can be applied. Note that the derivation of Figure 4.9 is left unfinished with the same triple to prove than the one it started with: we can copy/paste this derivation infinitely many times, making it infinite. This way of reasoning is called coinduction (we have already discussed about it in Section 2.1.2), and is accepted by Coq.

We thus need to allow infinite abstract derivations. To this end, the abstract evaluation relation—written $\Downarrow^\sharp$—is obtained as the greatest fixed point of $\mathcal{F}^\sharp$. The soundness of this extension has been proven in Coq. More importantly, a coinductive approach allows analysers to use more techniques, such as *invariants*, to infer their conclusions. The snippet of Program 4.10 shows the definition of $\Downarrow^\sharp$ in Coq. Note the symmetry between this definition and the concrete definition of $\Downarrow$ in Program 4.2; the only notable difference being in the constructor `aeval_cons`: $\Downarrow^\sharp$ applies all the rules which apply, and not just one.

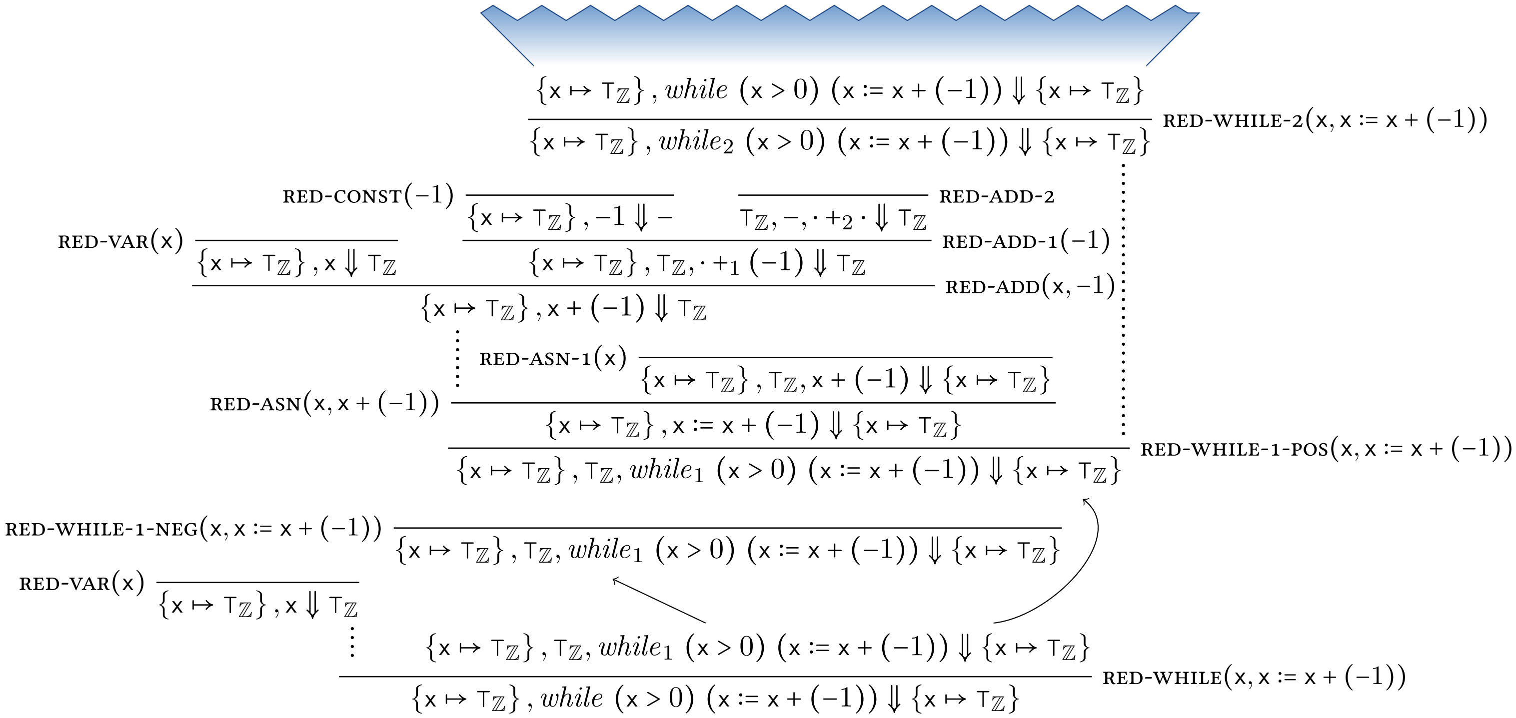


Figure 4.9: An infinite abstract derivation related to finite concrete derivations

```
CoInductive aeval : ast → term → ares → Prop :=
  | aeval_cons : forall sigma t r,
      (forall n,
        t = left n →
        acond n sigma →
        aapply n sigma r) →
      aeval sigma t r
with aapply : name → ast → ares → Prop :=
  | aapply_Ax : forall n ax sigma r,
      rule_struct n = Rule_struct_Ax _ →
      arule n = Rule_Ax ax →
      ax sigma = Some r →
      aapply n sigma r
  | aapply_R1 : forall n t up sigma sigma' r,
      rule_struct n = Rule_struct_R1 t →
      arule n = Rule_R1 _ up →
      up sigma = Some sigma' →
      aeval t sigma' r →
      aapply n sigma r
  | aapply_R2 : forall n t1 t2 up next
                       sigma sigma1 sigma2 r r',
      rule_struct n = Rule_struct_R2 t1 t2 →
      arule n = Rule_R2 up next →
      up sigma = Some sigma1 →
      aeval t1 sigma1 r →
      next sigma r = Some sigma2 →
      aeval t2 sigma2 r' →
      aapply n sigma r'.
```

Program 4.10: Coq definition of the abstract semantics $\Downarrow^\sharp$

4.4.3 Soundness of the Abstract Semantics

We have defined in Section 4.4.1.2 the local soundness as the conjunction of the soundness of the side-condition predicate $cond^\sharp$ with respect to $cond$ and the soundness $\sim$ of the transfer functions. We proved in Coq that under the local soundness, the concrete and abstract evaluation relations, $\Downarrow = lfp(\mathcal{F})$ and $\Downarrow^\sharp = gfp(\mathcal{F}^\sharp)$, are related as follows.

Theorem 4.1 (Soundness). *Let $t \in term$, $\sigma \in st$, $\sigma^\sharp \in st^\sharp$, $r \in res$ and $r^\sharp \in res^\sharp$.*

$$\text{If} \begin{cases} \sigma \in \gamma\left(\sigma^\sharp\right) \\ \sigma, t \Downarrow r \\ \sigma^\sharp, t \Downarrow^\sharp r^\sharp \end{cases} \quad \text{then } r \in \gamma\left(r^\sharp\right).$$

This theorem states that an abstract semantics can not miss any concrete (terminating) behaviours: once an abstract triple $\sigma^\sharp, t \Downarrow^\sharp r^\sharp$ has been derived in the abstract semantics, then any concrete triple $\sigma, t \Downarrow r$ starting from the same term t and a related semantic context σ (that is, $\sigma \in \gamma\left(\sigma^\sharp\right)$) has a result corresponding to the abstract result: $r \in \gamma\left(r^\sharp\right)$.

The Coq proof is detailed in its general case in Section 5.3. It proceeds by induction over the concrete derivation, recognising it step by step in the abstract derivation. The invariants are locally preserved by local soundness. This is exactly how Schmidt proves soundness: by defining abstract derivations so that they cover the concrete derivations. More precisely, Schmidt defines an inclusion relation between derivations and proves concrete derivations to be included in the corresponding abstract derivations (see Section 3.3).

We now present a sketch of the proof. As in the Coq proof, we proceed by induction over the concrete derivation. Let $\mathfrak{r}$ be the rule taken by the concrete derivation. By definition, the concrete side-condition $cond_{\mathfrak{r}}(\sigma)$ applies. By the soundness of side-conditions (Criterion 4.2), so does the abstract side-condition $cond^\sharp_{\mathfrak{r}}\left(\sigma^\sharp\right)$. Rule $\mathfrak{r}$ thus also applies in the abstract derivation: we are left to prove that $r \in \gamma\left(r^\sharp\right)$ when $\sigma \in \gamma\left(\sigma^\sharp\right)$, $(\sigma, t, r) \in apply_{\mathfrak{r}}(\Downarrow)$, and $\left(\sigma^\sharp, t, r^\sharp\right) \in apply_{\mathfrak{r}}\left(\Downarrow^\sharp\right)$. For instance, we consider the case in which Rule $\mathfrak{r}$ is of format 1. By Definition 4.1 of $apply$, we have the two equalities $up(\sigma) = \texttt{Some}(\sigma')$ and $up^\sharp\left(\sigma^\sharp\right) = \texttt{Some}\left(\sigma'^\sharp\right)$, as well as two reductions $\sigma', u_{1,\mathfrak{r}} \Downarrow r$ and $\sigma'^\sharp, u_{1,\mathfrak{r}} \Downarrow r^\sharp$. By the soundness of transfer functions (Criterion 4.5), we get $\sigma' \in \gamma\left(\sigma'^\sharp\right)$. We conclude $r \in \gamma\left(r^\sharp\right)$ by the induction hypothesis applied on $\sigma', u_{1,\mathfrak{r}} \Downarrow r$ and $\sigma'^\sharp, u_{1,\mathfrak{r}} \Downarrow r^\sharp$. The other cases (axioms and format 2 rules) are similar.

This theorem only provides guarantees for terminating results. For instance, there is no concrete derivation for the looping program $while\ (42 > 0)\ skip$. As abstract derivations are defined coinductively, and thus infinite derivations are accepted, it is possible to define an abstract semantics for this program. Figure 4.10a shows how we can infer $\bot$ as a result; this derivation is infinite: the unfinished part loops back to the beginning of the derivation

$$
\dfrac{\dfrac{}{42, \mathsf{x} \Downarrow +}\text{\textsc{red-const}}(42) \quad \dfrac{\begin{array}{c}\dfrac{}{\{\mathsf{x}\mapsto -\}, \mathit{skip} \Downarrow \{\mathsf{x}\mapsto -\}}\text{\textsc{red-skip}}\\ \vdots\end{array} \quad \dfrac{\{\mathsf{x}\mapsto -\}, \mathit{while}\ (42>0)\ \mathit{skip} \Downarrow \bot}{\{\mathsf{x}\mapsto -\}, \mathit{while}_2\ (42>0)\ \mathit{skip} \Downarrow \bot}\text{\textsc{red-while-2}}(42, \mathit{skip})}{\{\mathsf{x}\mapsto -\}, +, \mathit{while}_1\ (42>0)\ \mathit{skip} \Downarrow \bot}\text{\textsc{red-while-1-pos}}(42, \mathit{skip})}{\{\mathsf{x}\mapsto -\}, \mathit{while}\ (42>0)\ \mathit{skip} \Downarrow \bot}\text{\textsc{red-while}}(42, \mathit{skip})
$$

(a) With $\bot$ as a result

$$
\dfrac{\begin{array}{c}\dfrac{}{42, \mathsf{x} \Downarrow +}\text{\textsc{red-const}}(42)\\ \vdots\end{array} \quad \dfrac{\begin{array}{c}\dfrac{}{\{\mathsf{x}\mapsto -\}, \mathit{skip} \Downarrow \{\mathsf{x}\mapsto -\}}\text{\textsc{red-skip}}\\ \vdots\end{array} \quad \dfrac{\{\mathsf{x}\mapsto -\}, \mathit{while}\ (42>0)\ \mathit{skip} \Downarrow \{\mathsf{x}\mapsto +, \mathsf{y}\mapsto 0\}}{\{\mathsf{x}\mapsto -\}, \mathit{while}_2\ (42>0)\ \mathit{skip} \Downarrow \{\mathsf{x}\mapsto +, \mathsf{y}\mapsto 0\}}\text{\textsc{red-while-2}}(42, \mathit{skip})}{\{\mathsf{x}\mapsto -\}, +, \mathit{while}_1\ (42>0)\ \mathit{skip} \Downarrow \{\mathsf{x}\mapsto +, \mathsf{y}\mapsto 0\}}\text{\textsc{red-while-1-pos}}(42, \mathit{skip})}{\{\mathsf{x}\mapsto -\}, \mathit{while}\ (42>0)\ \mathit{skip} \Downarrow \{\mathsf{x}\mapsto +, \mathsf{y}\mapsto 0\}}\text{\textsc{red-while}}(42, \mathit{skip})
$$

(b) With unexpected result

Figure 4.10: Infinite abstract derivations for a looping program

over and over again. The result $\bot$ is the most precise result (the smallest in the considered poset) which can be expected: as $\gamma(\bot) = \varnothing$, we know by Theorem 4.1 that no concrete derivation can be built from this term and a corresponding semantic context.

Figure 4.10a is not the only abstract derivation which can be built from these term and semantic context: Figure 4.10b shows another derivation. We have mentioned in Section 3.1 that abstract semantics can be non-deterministic: one source of non-determinism is that several invariants can be found and proven for a given loop. In the case of Figure 4.10b, we have produced the alternative result $\{\mathsf{x} \mapsto +, \mathsf{y} \mapsto 0\}$. This result can be surprising, as none of the variables x and y are touched by the considered program. The theorem 4.1 is however not broken, as it is not possible to build any concrete derivation. It is thus important to have in mind the fact that only terminating behaviours are taken into account when interpreting the results of analyses defined in this dissertation.

Infinite loops are not the only ways to stop the construction of a concrete derivation: an execution may end being stuck. This can happen when mistakes are made in the definition of the concrete semantics—for instance if Rule red-if-1-pos(s_1, s_2) would have been replaced by Rule broken-if-1-pos(s_1, s_2) below. Its expected semantic context (E, v) does not match the statement s_1, and no rule apply on the triple $E, v, s_1 \Downarrow r$. It has already occurred during this thesis to get $\bot$ as the result of an analysis where it should not; after careful analysis, it appeared that the error was actually in the concrete semantics: the theorem held, but the analyser was analysing a broken concrete semantics. Section 4.5 proposes a way to avoid such issues when defining semantics.

$$\textsc{broken-if-1-pos}(s_1, s_2) \quad \frac{E, v, s_1 \Downarrow r}{E, v, \mathit{if}_1\ s_1\ s_2 \Downarrow r} \quad v > 0$$

Here follows the Coq version of Theorem 4.1. It has been proven in a completely parametrised way with respect to the concrete and abstract domains, as well as the rules: it can be instantiated for any (pretty-big-step) semantics.

```
Theorem soundness : forall t asigma ar,
  aeval _ _ _ t asigma ar →
  forall sigma r,
    gst asigma sigma → eval _ t sigma r → gres ar r.
```

The predicates `aeval` and `eval` respectively represent $\Downarrow^\sharp$ and $\Downarrow$ (see programs 4.2 and 4.10), and `gst` and `gres` are the respective concretisation functions for semantic contexts and results. The hypothesis which was expected to appear in Theorem 4.1 but did not is the exhaustivity of the semantics; we now discuss this hypothesis.

4.4.4 Exhaustivity

As stated in Section 4.3.1, the exhaustivity (also called *fullness*) of a semantics corresponds to the fact that all transfer functions are defined given a semantic context which satisfies the side-condition $cond^\sharp$. This is defined in Coq in Program 4.11. The predicate `applies` states whether a given semantic context σ applies on a given rule; that is, if the transfer functions are defined for this semantic context. Note how the definition of the $next$ transfer function is stated Line 10: once the semantic context σ is chosen, any result has to be accepted. Intuitively, although the type of the $next$ transfer function is `st → res → option st`, it should really be understood as `st → option (res → st)`.

This hypothesis is not needed to prove the soundness of the abstract semantics (Theorem 4.1). One way of understanding why is by examining closely the definition of $apply$, typically in Programs 4.2 (Line 11 for instance) and 4.10 (Line 12): in order to build a derivation—either a concrete or an abstract—the transfer functions have to be defined on the considered semantic contexts. If either the concrete or the abstract derivation can not be defined, then Theorem 4.1 does not apply. This section discusses the issues of inexhaustivity. In particular, it shows that inexhaustive abstract semantics are not problematic, but inexhaustive concrete semantics usually are.

In the abstract world, inexhaustive semantics are not an issue. Indeed, if a transfer function of an abstract rule fails but the side-condition states that the rule applies, then the construction of the abstract derivation fails. For instance, if we change the side-condition

```
Inductive applies : Rule → st → Prop :=
  | applies_Ax : forall sigma ax,
    (exists r, ax sigma = Some r) →
    applies (Rule_Ax ax) sigma
  | applies_R1 : forall sigma up,
    (exists sigma', up sigma = Some sigma') →
    applies (Rule_R1 _ up) sigma
  | applies_R2 : forall sigma up next,
    (exists sigma', up sigma = Some sigma') →
    (forall r, exists sigma', next sigma r = Some sigma') →
    applies (Rule_R2 up next) sigma.

Definition semantics_exhaustive := forall n sigma,
  cond n sigma →
  applies (rule n) sigma.
```

Program 4.11: Definition of the exhaustivity in Coq

of Rule red-var(x) to always apply, even when the semantic context is not even in the expected form, this side-condition trivially respects Criterion 4.2. Such a rule would however not be practical as its abstract transfer function is not always defined. As a consequence, it will not be possible to build most triples from the abstract immediate consequence $\mathcal{F}^\sharp$ as this abstract rule would always fire. However, all the semantic triples successfully built from such a semantics would be sound. If the abstract semantic is not exhaustive, but that the construction of an abstract derivation does not fail, this means that the incomplete abstract transfer functions were not needed out of their domain. Theorem 4.1 applies on such a derivation. This is very practical for development, as it means that an abstract semantic with "transfer functions left to be defined" is still valid. In particular, we can run the certified analysers of Section 4.6 even if some transfer functions are not yet implemented, as soon as they are not needed for the considered derivation. If the remaining parts are needed to build a derivation, analyses will fail without claiming wrong statements.

We now consider inexhaustivity in concrete semantics. Imagine that, because of a mistake, the Coq implementation of Rule red-var(x) only works when the value of x is positive in the given environment. Such an implementation is not exhaustive: the side-condition of Rule red-var(x) checks whether the semantic context is an environment, and whether it defines x, not whether it defines x to a positive value. From such a concrete semantic, the abstract transfer function $ax^\sharp$ constant to $+_0$ respects Criterion 4.4 about axiom transfer function: whenever the concrete function transfer is defined (that is in this example, when x is positive), it returns a positive value. The theorem only applies on the concrete and abstract derivations which can be built. If the inexhaustivity of the concrete rule was a programming error, then the abstract semantics built from this inexhaustive concrete semantics may miss concrete results. A similar issue happens when the concrete semantics can be stuck, for instance because of Rule broken-if-1-pos(s_1, s_2) of previous section. As can be seen in these examples, inexhaustive concrete semantics are prone to mistakes.

$$\begin{aligned} st\,(e) &= Env \\ st\,(s) &= Env \\ st\,(\cdot +_1 e_2) &= Env \times Out_e \\ st\,(\cdot +_2 \cdot) &= Val \times Out_e \end{aligned}$$

$$\begin{aligned} st\,(\mathsf{x} :=_1 \cdot) &= Env \times Out_e \\ st\,(\cdot ;_1 s_2) &= Out_s \\ st\,(\mathit{if}_1\, s_1\, s_2) &= Env \times Out_e \\ st\,(\mathit{while}_1\, e\, s) &= Out_s \\ st\,(\mathit{while}_2\, e\, s) &= Env \times Out_e \end{aligned}$$

(a) Semantic contexts

$$\begin{aligned} res\,(e_x) &= Out_e \\ res\,(s_x) &= Out_s \\ res\,(e) &= Out_e \\ res\,(s) &= Out_s \end{aligned}$$

(b) Results

Figure 4.11: Definition of the (dependent) types for semantic contexts and results

To summarize, inexhaustive concrete semantics are to be avoided as they prevent concrete derivations to be built, as well as Theorem 4.1 to apply on these missing derivations. Inexhaustive abstract semantics are however nevertheless sound. Their inexhaustivity may still prevent some abstract derivation to be built: the best possible result might not be derivable in inexhaustive abstract semantics. In particular, they might not be able to provide a result in some cases. But their results will always be sound.

4.5 Dependently Typed Pretty-big-step

Rule BROKEN-IF-1-POS(s_1, s_2) defined in Section 4.4.3 shows how untyped our formalisation is. The type of the semantic context should depend on the term being evaluated. This remark is the starting point of the alternative specification presented in this section. Although more principled, it was not implemented in Coq because of difficulties with Coq's dependent types. This section presents this formalisation alternative, but none of the (re)definition made in this section will apply to the rest of the dissertation.

In this setting, the types of semantic contexts and results depends on the current term t. For instance expressions return expression results. Figure 4.11 presents the concrete definition of these types; they are compatible with the rules of Figures 3.4 and 4.2, but not with Rule BROKEN-IF-1-POS(s_1, s_2). The syntactic part is left unchanged: semantics carry a set $\mathcal{N}$ of rule names, each rule $\mathfrak{r}$ carries a term $\mathfrak{l}_\mathfrak{r}$, a rule format $\mathit{kind}\,(\mathfrak{r})$, as well as some additional terms $\mathfrak{u}_1$, $\mathfrak{u}_2$, and $\mathfrak{n}_2$ depending on their format. The only difference on the syntactic part is that the additional terms carry proofs that their corresponding rule is on the right format, as shown below. This enforces for instance that $\mathfrak{n}_2$ can only be applied on a format 2 rule. These terms influence the different types of the semantic aspects of rules.

$$\begin{aligned} \mathfrak{u}_1 &: (\mathfrak{r} \in \mathcal{N}) \to (\mathit{kind}\,(\mathfrak{r}) = R_1) \to \mathit{term} \\ \mathfrak{u}_2 &: (\mathfrak{r} \in \mathcal{N}) \to (\mathit{kind}\,(\mathfrak{r}) = R_2) \to \mathit{term} \\ \mathfrak{n}_2 &: (\mathfrak{r} \in \mathcal{N}) \to (\mathit{kind}\,(\mathfrak{r}) = R_2) \to \mathit{term} \end{aligned}$$

The predicate *cond* takes a semantic context whose type depends on the term on which its rule applies. In particular, the side-condition no longer has to check whether the semantic context is in the expected form: the side-condition of Rule RED-CONST(c) can simply be $True$, the condition on the semantic context being enforced by its type.

$$cond : (\mathfrak{r} \in \mathcal{N}) \to st\,(\mathfrak{l}_{\mathfrak{r}}) \to Prop$$

Transfer functions are now dependently typed. As a consequence, the type *Rule* is now parametrised by the name of the rule. The *rule* function has thus the following type.

$$rule : (\mathfrak{r} \in \mathcal{N}) \to Rule_{\mathfrak{r}}$$

Dependent types are expressive enough to enable transfer functions to only be applied when their corresponding side-condition applies. This removes the need for transfer functions to be partial. As a consequence, semantics are exhaustive by definition in this setting: the transfer functions are enforced to be defined by their type.

- An axiom rule $\mathfrak{r}$ has a transfer function *ax* of the following type. It is a total function, but requires a proof that the rule applies as a parameter.

  $$ax : (\sigma \in st\,(\mathfrak{l}_{\mathfrak{r}})) \to cond_i\,(\sigma) \to res\,(\mathfrak{l}_{\mathfrak{r}})$$

  Some rules—Rule RED-CONST(c) for instance—will ignore the proof argument, but its presence still guarantees the transfer function to only be applied when the rule applies. Other rules, such as Rule RED-VAR$(\mathtt{x})$, will use this proof to access a particular part of the semantic context: the proof of $\mathtt{x} \in dom\,(E)$ can be used to access the environment E—as required by some Coq libraries, including TLC [Cha10].
- Format 1 rules are similar to axioms, with a transfer function of the following type.

  $$up : (\sigma \in st\,(\mathfrak{l}_{\mathfrak{r}})) \to cond_{\mathfrak{r}}\,(\sigma) \to st\,(\mathfrak{u}_{1,\mathfrak{r}})$$

  Format 1 rules also come with an additional constraint: the result of the premise of format 1 rules is propagated as-is (see Figure 4.5). These two results are thus required to have the same type.

  $$kind\,(\mathfrak{r}) = R_1 \to res\,(\mathfrak{u}_{1,\mathfrak{r}}) = res\,(\mathfrak{l}_{\mathfrak{r}}) \tag{4.7}$$

- Format 2 rules start similarly to format 1 rules. The *next* transfer function is no longer partial thanks to the proof of the rule application given as argument.

  $$\begin{aligned} up &: (\sigma \in st\,(\mathfrak{l}_{\mathfrak{r}})) \to cond_{\mathfrak{r}}\,(\sigma) \to st\,(\mathfrak{u}_{2,\mathfrak{r}}) \\ next &: (\sigma \in st\,(\mathfrak{l}_{\mathfrak{r}})) \to cond_{\mathfrak{r}}\,(\sigma) \to res\,(\mathfrak{u}_{2,\mathfrak{r}}) \to st\,(\mathfrak{n}_{2,\mathfrak{r}}) \end{aligned}$$

Format 2 rules are also associated a constraint about result types being equal, their last result being propagated in similar fashion than format 1 rules.

$$kind\left(\mathfrak{r}\right) = R_2 \rightarrow res\left(\mathfrak{n}_{2,\mathfrak{r}}\right) = res\left(\mathfrak{l}_{\mathfrak{r}}\right) \tag{4.8}$$

The rest of the formalisation naturally follows from these changes. The resulting formalisation is very elegant, but its implementation in Coq proved to be quite challenging. The typical difficulty with these dependent types follows from the constraints 4.7 and 4.8. These constraints force result types to be the same; however, as usually with dependent types, a lot of predicates require these terms to have a specific (syntactical) type. For instance, Coq will defensively reject the result given to a *next* transfer function if the type of this result is not syntactically $res\left(\mathfrak{u}_{2,\mathfrak{r}}\right)$. Rewriting under such results requires the usage of heterogeneous, or "John Major's", equality [McB02]; which can be really painful to use when such syntactic constraints apply on types.

The dependently typed formalisation was thus simplified for the sake of the Coq formalisation: the type of semantic contexts st (respectively results res) is the union of every semantic context types (respectively every results types). Zooming out on what Section 3.1 explained, we have described how we can define and prove sound an abstract semantics; the next step consists in building analysers based on this abstract semantics.

4.6 Building Certified Analysers

Now that abstract semantics are defined as functional data structures, it is possible to build some automatic definitions on top of them. We shall focus in this section on how to build generic analysers from abstract semantics. The abstract semantics $\Downarrow^\sharp$ is the set of all triples provable using abstract inference rules. From a program t and an abstract semantic context $\sigma^\sharp$, the smallest $r^\sharp$ such that $\sigma^\sharp, t \Downarrow^\sharp r^\sharp$ (when it exists—the restrictions of Section 3.2.4 no longer enforcing its existence) corresponds to the most precise result returned by an analysis; it is, however, rarely computable. Designing a certified analysis amounts to write a program which returns a result accepted by the abstract semantics.

There are several ways to define such analysers and prove them sound. A sound-by-construction analyser would be an analyser returning an abstract derivation as a result, but defining such an analyser would be unpractical. Another way is to separately define and prove analysers. To this end, we heavily rely on the coinductive definition of $\Downarrow^\sharp$ to prove the soundness of static analysers. In order to prove that a given analyser $\mathcal{A} : st^\sharp \rightarrow term \rightarrow res^\sharp$ is sound with respect to $\Downarrow^\sharp$—and thus with respect to the concrete semantics by Theorem 4.1—, it is sufficient to define for every term t and semantic context $\sigma^\sharp$ a set $R \in \mathcal{P}\left(st^\sharp \times term \times res^\sharp\right)$ such that $\left(\sigma^\sharp, t, \mathcal{A}\left(\sigma^\sharp, t\right)\right) \in R$ and prove it *coherent*, that is $R \subseteq \mathcal{F}^\sharp\left(R\right)$. This is called Park's principle [Par69], and is a general way

```
Inductive aeval_f : ast → term → ares → Prop :=
  | aeval_f_cons : forall (R : ast → term → ares → Prop) sigma t r,
    (forall sigma t r,
       R sigma t r →
       aeval_check R sigma t r) →
    R sigma t r →
    aeval_f sigma t r.
```

Program 4.12: Alternative Coq definition of $\Downarrow^\sharp$

of proving coinductive constructions such as $\Downarrow^\sharp$. This approach is sound as $\Downarrow^\sharp$ is defined as the greatest fixed point of $\mathcal{F}^\sharp$ (see Section 4.4.2.2): by Tarski's theorem, the greatest fixed point is the union of all set R such that $R \subseteq \mathcal{F}^\sharp(R)$ [Tar55].

We instantiate this principle in Coq in Program 4.12 through the alternative definition `aeval_f` of $\Downarrow^\sharp$. The parametrised predicate `aeval_check` applies one step of the reduction: it exactly corresponds to $\mathcal{F}^\sharp$ and is defined in Coq similarly to `aeval` (Program 4.10). More precisely, `aeval` is the coinductive closure of `aeval_check`—we could not define it directly as-is because Coq's coinduction relies on some syntactic productivity checks, which would not be fulfilled by a direct definition. We thus require an intermediary set `R` and a proof of its coherence with respect to `aeval_check`. The following equivalence theorem enables us to use Park's principle.

```
Theorem aevals_equiv : forall t sigma r,
  aeval t sigma r ↔ aeval_f t sigma r.
```

Using this principle, we have built and proved the soundness of several different analysers; they are available in the Coq files accompanying this dissertation [Bod16]. Most of these analysers are generic and can be reused as-is with any abstract semantics built using our framework. We now describe three of such analysers:

- Admitting Rule ABS-TOP as a trivial analyser which always returns ⊤ independently of the given term and semantic context.
- Building a certified program verifier able to check loop invariants given by an (unverified) oracle and to use them to make abstract interpretations of programs.
- Building flat analysers from a concrete semantics.

4.6.1 Trivial Analyser

We have mentioned that Rule ABS-TOP of Section 4.2 is a useful rule often taken for granted: it enables an analyser to abort the analysis of a part of a derivation and to continue the analysis on the rest of the derivation. For instance, this rule is applied in many JavaScript analysers when encountering a `eval`-construct: potentially anything can happen, but the rest of the program may catch pathological behaviours. This rule is not part of the abstract semantic presented in Section 4.4, but it follows from the rest of the semantics. To prove this rule sound, we need three hypotheses. First, the abstract poset of results obviously needs a greatest element $\top_{res^\sharp}$. Second, the abstract semantics has to be exhaustive—this property is not always fulfilled, but is common (see Section 4.4.4). Third, we need a weakening rule such as Rule GLUE-WEAKEN of Section 5.1.1—we temporary admit it.

$$\textsc{glue-weaken}\quad \frac{\sigma^\sharp \sqsubseteq \sigma^{\sharp\prime} \qquad \sigma^{\sharp\prime}, t \Downarrow r^{\prime\sharp} \qquad r^{\prime\sharp} \sqsubseteq r^\sharp}{\sigma^\sharp, t \Downarrow r^\sharp}$$

Admitting Rule ABS-TOP exactly amounts to prove that the corresponding trivial analyser is sound. We define the set $\Downarrow^\sharp_\top = st^\sharp \times term \times \{\top_{res^\sharp}\}$ and prove it coherent. We have to prove that every semantic triple $\left(\sigma^\sharp, t, \top_{res^\sharp}\right)$ is also part of $\mathcal{F}^\sharp\left(\Downarrow^\sharp_\top\right)$: for every rule $\mathfrak{r}$ which applies—that is, $cond^\sharp_\mathfrak{r}\left(\sigma^\sharp\right)$—then $\left(\sigma^\sharp, t, \top_{res^\sharp}\right) \in apply_\mathfrak{r}\left(\Downarrow^\sharp_\top\right)$. As $\top_{res^\sharp}$ is greater than any other result, we just have to prove that there exists at least one result $r^\sharp$ such that $\left(\sigma^\sharp, t, r^\sharp\right) \in apply_\mathfrak{r}\left(\Downarrow^\sharp_\top\right)$: Rule GLUE-WEAKEN can then weaken $r^\sharp$ into $\top_{res^\sharp}$. The existence of such an $r^\sharp$ is provided by the exhaustivity of the abstract semantics.

The trivial analyser shows that it is possible to prove ad-hoc rules using Park's principle. An example in which proving such specialised analysers can be useful is the rule scheme RANDOM(c) (defined for all $c \in \mathbb{N}$) below, applying to a term $random$. We can similarly prove that the abstract Rule ABS-RANDOM is a valid abstraction. It will furthermore be much easier to prove as this rule is not recursive.

$$\textsc{Random}(c)\quad \frac{}{E, random \Downarrow c} \qquad\qquad \textsc{Abs-Random}\quad \frac{}{E, random \Downarrow \top_\mathbb{Z}}$$

4.6.2 Certified Program Verifier

To enable the usage of external heuristics to provide potential program properties, and thus relax proof obligations, we have also proved a verifier: it takes a set of triples $O \in \mathcal{P}\left(st^\sharp \times term \times res^\sharp\right)$—which we call an *oracle*—and accepts or rejects it. An acceptance implies the soundness of every triple of O (but the converse does not hold). The verifier proceeds as follows. For every triple $o = \left(\sigma^\sharp, t, r^\sharp\right) \in O$, the verifier checks that it can be deduced from finite derivations; these derivations are allowed to stop the computation when reaching an element of O. In other words, the verifier checks that $O \subseteq \mathcal{F}^{\sharp^+}(O)$.

In practice, derivations are built backwards: the verifier computes hypotheses implying the considered triple o—a subset S of $\mathcal{F}^{\sharp^{-1}}(o)$ such that $o \in \mathcal{F}^{\sharp^+}(S)$. It then recursively iterates on S until it reaches only elements of O, or until it gives up. During this iteration, axioms may be encountered: axioms have no premises and thus reduce the size of the considered S. This is illustrated in Figure 4.12. We proved the following theorem.

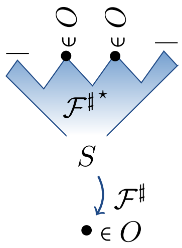


Figure 4.12: An illustration of the action of the verifier

Theorem 4.2 (Soundness of the verifier). *If the verifier accepts the oracle O, then O is coherent: $O \subseteq \mathcal{F}^{\sharp^{+}}(O)$. This implies $O \subseteq \Downarrow^{\sharp}$.*

For this verifier to be extractable, we need to provide a key ingredient: a function computing the list of rules which apply to a given semantic context $\sigma^{\sharp}$ and term t. The verifier then forks between all applicable rules. There are several ways to compute such a list. One is to provide a computable version of $\mathfrak{l}^{-1}$—for each term t, give the rules which may apply—and prove the decidability of side-conditions. The verifier then filters from $\mathfrak{l}^{-1}(t)$ all the rules which apply. There are cases in which an infinite number of rules apply. For instance all concrete rules $\textsc{Random}(c)$ of the previous section apply on the term $random$. In such cases, it is not always possible to provide a (finite) list of applicable rules: the provided $\mathfrak{l}^{-1}$ is left partial and special analysers have to tackle the analysis of such terms.

As for JSRef (see Section 2.6.1), it is difficult to prove in Coq that the verification terminates—it often does not. To this end, the verifier is parametrised by a fuel and a decidable predicate `is_looping_branch` which indicates which terms may indefinitely loop (in this example, terms of the form $while_1\ s_1\ s_2$): each time the verifier encounters such a potentially looping term, the fuel is decremented. When the fuel reaches zero, the analysis aborts.

This program verifier has a very distinct structure than these of previously defined abstract interpreters written in Coq [CP10; Ber09]. The verifier is here written in its generality: it takes any abstract semantics as an argument, whereas traditional abstract interpreters are defined for a particular abstract semantics. It is also proven sound for all abstract semantics. Furthermore, this verifier does not solve any fixed points: it takes as an argument an oracle O, which will be used as a basis for the analysis. In particular, the oracle O can be computed using techniques such as widening and narrowing [CC77a]: heuristics to compute this invariant are not considered here to be part of the program verifier. The program verifier proposed here is actually closer to the abstract analysers generated by the $\mathbb{K}$ framework in its structure than to traditional abstract interpreters written in Coq—it however provides more guarantees, as it is proven sound in Coq.

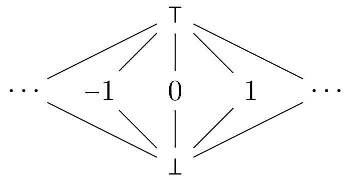


Figure 4.13: Hasse diagram of a flat domain

We extracted the verifier into OCAML and ran it on simple examples [Bod16]. Note that it can be given any oracle, possibly unsound. In the given examples, oracles were constructed by following abstract derivation trees up to a given number of loop unfoldings and ignoring deeper branches. Other oracles could have been used without trouble.

As an example, consider the following program which computes 6×7 using a while loop.

$$\mathsf{a} := 6; \mathsf{b} := 7; \mathsf{r} := 0; \mathsf{n} := \mathsf{a}; \mathit{while}\ (\mathsf{n} > 0)\ (\mathsf{r} := \mathsf{r} + \mathsf{b}; \mathsf{n} := \mathsf{n} + (-1))$$

From the empty abstract environment, our analyser returns the following result.

$$\left(\{\mathsf{r} \mapsto +, \mathsf{b} \mapsto +, \mathsf{a} \mapsto +, \mathsf{n} \mapsto \top_{\mathbb{Z}}\}, \overline{\mathit{err}}^{\sharp}\right)$$

The $\overline{\mathit{err}}^{\sharp}$ means that we successfully proved that the program does not abort (it does not access any undefined variable). We also deduce from this result that the returned value is strictly positive: the loop is executed at least once. Note that this is the best result we can get on this example given the abstract domains of Section 4.1 and the constraints on the abstract semantics. In particular, remark that the sign domain can not count how many times the loop needs to be unfolded, hence the abstract derivation is infinite. Nevertheless, the analysis deduces significant information. It is possible to get more precise results, in particular showing that n ends up being zero. Chapter 5 introduces Rule GLUE-TRACE-PARTITIONING, which enables such precise results.

4.6.3 Flat Analysers

We now consider flat domains instead of the domain of Section 4.1. A flat domain is exactly the concrete domain with two additional constructs $\top$ and $\bot$. Figure 4.13 shows its Hasse diagram: the order $\sqsubseteq$ is the minimal relation such that $\top$ is greater than all elements and $\bot$ smaller than all elements. The generic analysers of the previous section still apply—in fact, it is easy to define the abstract transfer functions on this domain in a generic way.

The resulting analyser is a flat analyser. It behaves like a concrete interpreter with two additional results: it may return $\bot$ on some looping programs, and returns $\top$ when more than two rules apply. There is thus no need of defining an interpreter for a concrete semantics in this formalism, as it is given for free: the side-conditions have to be proven decidable, and a computable function $\downharpoonleft^{-1}$ is required, but these are the only requirements.

$$
\begin{array}{llll}
e ::= c \in \mathbb{Z} & e_e ::= \cdot +_1 e & s \in \mathit{stat} ::= \mathit{skip} & s_e ::= \mathsf{x} :=_1 \cdot \\
\quad \mid \mathsf{x} \in \mathit{Var} & \quad \mid \cdot +_2 \cdot & \quad \mid s_1 ; s_2 & \quad \mid \cdot ;_1 s_2 \\
\quad \mid e_1 + e_2 & \quad \mid @_1 (e_2) & \quad \mid \mathsf{x} := e & \quad \mid \mathit{if}_1\, s_1\, s_2 \\
\quad \mid \lambda \mathsf{x}.s & \quad \mid @_2 & \quad \mid \mathit{if}\ (e > 0)\ s_1\, s_2 & \quad \mid \mathit{while}_1\ (e > 0)\ s \\
\quad \mid e_1\, (e_2) & \quad \mid @_3 & \quad \mid \mathit{while}\ (e > 0)\ s & \quad \mid \mathit{while}_2\ (e > 0)\ s \\
 & & \quad \mid \mathit{return}\, e & \quad \mid \mathit{return}_1 \cdot
\end{array}
$$

Figure 4.14: Updating the language of Figure 4.1

Theorem 4.1 still applies. Interestingly, it translates in this setting to a proof that the given concrete interpreter is complete: no result is missed by such an analyser. Such an analyser for the JSCert semantics could thus be a solution to establish the completeness discussed in Section 2.7.5. Instead of the flat domain, other more precise variants can be use, such as a powerset lattice. Applying this method with a powerset lattice would build an interpreter computing what is called the collecting semantics. It is the most precise analysis which can be performed on a program, but it requires so much resources (there can be a huge number of program states) that it is of little practical use in this setting.

4.7 Evaluation

We have defined a framework which lets us define in Coq pretty-big-step concrete semantics and abstract them in a guided way. The framework then provides a way to prove the abstract semantics sound by only proving local properties, considerably reducing the amount of proof to be done. The framework also provides some generic analysers, which can be extracted in OCaml and run. We now examine how this framework behaves when the concrete semantics changes, in particular about the associated proof effort. We conclude by considering which abstract rules of Section 4.2 are captured by the framework.

4.7.1 Extending a Semantics

To check whether our requirements are indeed local, we update the concrete semantics to add first-class functions in our language. We now consider how much work is needed to update the certified analysers of section 4.6. The new syntax of the language is defined in Figure 4.14. The expression $\lambda \mathsf{x}.s$ defines a function (for simplicity, we only consider functions with one argument). As for JavaScript, a function executes a statement, whose execution should end with a $\mathit{return}\, e$ statement; the expression $e_1\, (e_2)$ represents a function call. These new terms come with their own intermediary terms.

When calling a function, a local environment called context $C \in \mathit{Env}$ is created to carry the value of the function argument. When creating a closure with the expression $\lambda \mathsf{x}.s$, the current context is stored. To implement these environments, we use a structure close to Java-

Script's declarative environments records (see Section 1.2.3.1): environments are stored in a global heap H_e of environments. The heap H_e maps environment locations $\ell_e, \ell_c \in \mathcal{L}_e$ to environments $E \in Env = Var \rightharpoonup_{fin} Val$. The function *fresh* takes a heap H_e as an argument and returns a fresh location $\ell_e \notin dom(H_e)$.

The concrete domains have been updated from Definition 3.1. Program states are now composed of three different components:

- The global heap $H_e : \mathcal{L}_e \rightharpoonup_{fin} Env$.
- An environment location ℓ_e pointing to the global environment E.
- An environment location ℓ_c pointing to a context $C \in Env$. This context carries the local scope of the current function call.

The set Val of values now includes both basic values in $\mathbb{Z}$ and closures $(\ell_c, \lambda x.s)$. There are now three kinds of result which a statement can return: a normal result (a state), an error, or a return result, which is a triple of an environment heap H_e, the location of the current global environment ℓ_e, and a value v. We write the latter $ret(H_e, \ell_e, v)$. Expressions can now also alter the current environment heap through function calls: expressions now return either an error or a triple of an environment heap H_e, a location to the global environment ℓ_e, and a value. To simplify notations, we consider that each environment location ℓ_e associated with an environment heap H_e is in the domain of this heap. This invariant is conserved by the semantics: environment locations are never removed from the environment heap H_e. We could add a side-condition $\ell_c \in dom(H_e)$ in rules such as Rule red-var-local to make this constraint explicit.

Updating program states does not invalidate the Coq definition of most transfer functions: the inferred type of some monads changed, but most definitions are left unchanged. The **abort** predicate has to be changed to catch results of the form $ret(H_e, \ell_e, v)$. Furthermore, the new Rule red-app-3-ret catches these results: the aborting Rule red-error-expr(e) now features an **intercept** predicate defined in Figure 4.16. This is similar to Rule red-expr-abort of JSCert (see Figure 2.5).

Rules red-var(x), red-var-undef(x), and red-asn-1(x), which access environments have to be updated. Furthermore, the rules manipulating expression results also have to be updated, as the type of these results changed. For most of these rules, the changes are minor, but the effort could probably be further reduced by applying to this formalism some meta-theory frameworks [DSS13]. There are now three rules to access variables: the variable can be in the local environment, the global environment, or could be undefined. Figure 4.15 shows the rules which have been updated, and Figure 4.17 shows the additional rules which have been added to manipulate functions. Note the behaviour of contexts when being updated by Rule red-var-local(x): the changes are only visible in the scope of the current function, but not propagated to the scope of eventual enclosing functions. This choice has been made to keep the semantics simple. Both the syntactic (the set of rule names, the function $kind$, and the terms $\mathfrak{l}$, $\mathfrak{u}_1$, $\mathfrak{u}_2$, and $\mathfrak{n}_2$) and semantic (side-conditions and transfer functions) aspects of the added rules have to be created, but apart from the expected rules of Figure 4.15, no rules have to be changed from the previous semantics. Note how Rule RED-APP-2(s) takes the statement s as a parameter in its name: syntactic aspects of rule should not depend on the semantic domain; this is a problem for function calls, but we solve this by adding the missing information in the rule name.

The abstract semantics have to be updated, but only the parts which have been changed in the concrete semantics suffer changes in the abstract semantics: only the newly added rules and the rules which changed in the concrete semantics are changed. There is no stack in the concrete semantics. Indeed, the stack is hidden in the derivation structure: when a function call ends, the global environment is returned, but the computation continues in the local environment from where it called the function. This makes the abstraction quite straightforward. The new abstraction for values is shown below. Environments locations $\ell_e \in \mathcal{L}_e$ are abstracted by abstract environment locations $\eta \in \mathcal{L}_e^\sharp$. The environment heap H_e is abstracted by an abstract heap $H_e^\sharp : \mathcal{L}_e^\sharp \to Env^\sharp$. Section 6.3.2 provide more details on how environments are abstracted.

$$\begin{aligned}
Val^\sharp &= Sign^\sharp \times \mathcal{C}^\sharp \\
Store^\sharp &= \left(Val^\sharp + undef^\sharp \right)^\top \\
\mathcal{C}^\sharp &= \mathcal{P}\left(Var \times Stat \times \mathcal{L}_e^\sharp \right) \\
Env^\sharp &= Var \to Store^\sharp
\end{aligned}$$

As expected, the proof of local soundness needed for Theorem 4.1 has to be updated; but the local proofs of soundness of the unchanged rules is still accepted by Coq: only the newly added rules have to be proven locally sound. This justifies the adjective "local" for the local soundness, as local changes in the semantics only yield local changes in the soundness proof, and in the expected places.

At this stage, we already have an abstract semantics proven sound, but we may want to also update the analysers, such as the one defined in Section 4.6.2. To this end, we have to update the function $\mathfrak{l}^{-1}$ providing the set of potentially applicable rules: the term x returns the additional rule RED-VAR-LOCAL(x) in its associated list, and the new terms of Figure 4.14 return their associated list. Interestingly, it not possible to return any list for the term $@_3$: to return the rule name RED-APP-2(s), we need to know the statement s, which is in the environment E_f; this problem is fixed by defining a more general version of the $\mathfrak{l}^{-1}$ function taking the semantic context. Interestingly, the proof of decidability of side-conditions has not to be updated, as it is entirely taken care by type classes (see Section 3.4.1), leaving no effort from the user. The term $@_3$ is added to the terms recognised by `is_looping_branch`, as a potentially looping term (because of mutually recursive functions). These changes are actually enough to extract and run an analyser. Let us for instance consider the following program computing 6×7 using a function; to make the

$$\frac{}{H_e, \ell_e, \ell_c, c \Downarrow H_e, \ell_e, c}\ \textsc{red-const}(c)$$

$$\frac{}{H_e, \ell_e, \ell_c, \mathsf{x} \Downarrow H_e, \ell_e, \ell_c[\mathsf{x}]}\ \mathsf{x} \in dom\,(H_e[\ell_c])\quad \textsc{red-var-local}(\mathsf{x})$$

$$\frac{}{H_e, \ell_e, \ell_c, \mathsf{x} \Downarrow H_e, \ell_e, E[\mathsf{x}]}\ \mathsf{x} \in dom\,(H_e[\ell_e]) \wedge \mathsf{x} \notin dom\,(H_e[\ell_c])\quad \textsc{red-var-global}(x)$$

$$\frac{}{H_e, \ell_e, \ell_c, \mathsf{x} \Downarrow err}\ \mathsf{x} \notin dom\,(H_e[\ell_e]) \wedge \mathsf{x} \notin dom\,(H_e[\ell_c])\quad \textsc{red-var-undef}(x)$$

$$\frac{H_e, \ell_e, \ell_c, e_1 \Downarrow r \qquad \ell_c, r, \cdot +_1 e_2 \Downarrow r'}{H_e, \ell_e, \ell_c, e_1 + e_2 \Downarrow r'}\ \textsc{red-add}(e_1, e_2)$$

$$\frac{H_e, \ell_e, \ell_c, e_2 \Downarrow r \qquad v_1, r, \cdot +_2 \cdot \Downarrow r'}{\ell_c, (H_e, \ell_e, v_1), \cdot +_1 e_2 \Downarrow r'}\ \textsc{red-add-1}(e_2)$$

$$\frac{}{v_1, (H_e, \ell_e, v_2), \cdot +_2 \cdot \Downarrow H_e, \ell_e, v_1 + v_2}\ \textsc{red-add-2}$$

$$\frac{H_e, \ell_e, \ell_c, e \Downarrow r \qquad \ell_c, r, \mathsf{x} :=_1 \cdot \Downarrow r'}{H_e, \ell_e, \ell_c, \mathsf{x} := e \Downarrow r'}\ \textsc{red-asn}(\mathsf{x}, e)$$

$$\frac{\ell_e' = fresh\,(H_e) \qquad E = H_e[\ell_e]}{\ell_c, (H_e, \ell_e, v), \mathsf{x} :=_1 \cdot \Downarrow H_e[\ell_e' \leftarrow E[\mathsf{x} \leftarrow v]], \ell_e', \ell_c}\ \mathsf{x} \notin dom\,(H_e[\ell_c])\quad \textsc{red-asn-1}(\mathsf{x})$$

$$\frac{\ell_c' = fresh\,(H_e) \qquad C = H_e[\ell_c]}{\ell_c, (H_e, \ell_e, v), \mathsf{x} :=_1 \cdot \Downarrow H_e[\ell_c' \leftarrow C[\mathsf{x} \leftarrow v]], \ell_e, \ell_c'}\ \mathsf{x} \in dom\,(C)\quad \textsc{red-asn-1-local}(\mathsf{x})$$

$$\frac{H_e, \ell_e, \ell_c, e \Downarrow r \qquad \ell_c, r, if_1\, s_1\, s_2 \Downarrow r'}{H_e, \ell_e, \ell_c, if\ (e > 0)\ s_1\, s_2 \Downarrow r'}\ \textsc{red-if}(e, s_1, s_2)$$

$$\frac{H_e, \ell_e, \ell_c, s_1 \Downarrow r}{\ell_c, (H_e, \ell_e, v), if_1\, s_1\, s_2 \Downarrow r}\ v > 0\quad \textsc{red-if-1-pos}(s_1, s_2)$$

$$\frac{H_e, \ell_e, \ell_c, s_2 \Downarrow r}{\ell_c, (H_e, \ell_e, v), if_1\, s_1\, s_2 \Downarrow r}\ v \leqslant 0\quad \textsc{red-if-1-neg}(s_1, s_2)$$

$$\frac{H_e, \ell_e, \ell_c, e \Downarrow r \qquad \ell_c, r, while_1\ (e > 0)\ s \Downarrow r'}{H_e, \ell_e, \ell_c, while\ (e > 0)\ s \Downarrow r'}\ \textsc{red-while}(e, s)$$

$$\frac{}{\ell_c, (H_e, \ell_e, v), while_1\ (e > 0)\ s \Downarrow H_e, \ell_e, \ell_c}\ v \leqslant 0\quad \textsc{red-while-1-neg}(e, s)$$

$$\frac{H_e, \ell_e, \ell_c, C, s \Downarrow r \qquad r, while_2\ (e > 0)\ s \Downarrow r'}{\ell_c, (H_e, \ell_e, v), while_1\ (e > 0)\ s \Downarrow r'}\ v > 0\quad \textsc{red-while-1-pos}(e, s)$$

$$\frac{}{\sigma, e \Downarrow err}\ \mathbf{abort}\,\sigma \wedge \neg\mathbf{intercept}_e\,\sigma\quad \textsc{red-error-expr}(e)$$

Figure 4.15: Rules updated to account for the semantic changes

$$\frac{}{\mathbf{intercept}_{@_3}\ ret\,(H_e, \ell_e, v)}$$

Figure 4.16: The **intercept** predicate

$$\dfrac{}{H_e, \ell_e, \ell_c, \lambda \mathsf{x}.s \Downarrow H_e, \ell_e, (\ell_c, \lambda \mathsf{x}.s)}\ \textsc{red-lambda}(\mathsf{x}, s)$$

$$\dfrac{H_e, \ell_e, \ell_c, e_1 \Downarrow r \qquad \ell_c, r, @_1 (e_2) \Downarrow r'}{H_e, \ell_e, \ell_c, e_1 (e_2) \Downarrow r'}\ \textsc{red-app}(e_1, e_2)$$

$$\dfrac{H_e, \ell_e, \ell_c, e_2 \Downarrow r \qquad \ell_c', \mathsf{x}, s, r, @_2 \Downarrow r'}{\ell_c, \left(H_e, \ell_e, \left(\ell_c', \lambda \mathsf{x}.s\right)\right), @_1 (e_2) \Downarrow r'}\ \textsc{red-app-1}(e_2)$$

$$\dfrac{\ell_c' = \mathit{fresh}\,(H_e) \qquad C = H_e\,[\ell_c] \qquad H_e\left[\ell_c' \leftarrow C\,[\mathsf{x} \leftarrow v]\right], \ell_e, \ell_c', s \Downarrow r \qquad r, @_3 \Downarrow r'}{\ell_c, \mathsf{x}, s, (H_e, \ell_e, v), @_2 \Downarrow r'}\ \textsc{red-app-2}(s)$$

$$\dfrac{}{\mathit{ret}\,(H_e, \ell_e, v), @_3 \Downarrow H_e, \ell_e, v}\ \textsc{red-app-3-ret}$$

$$\dfrac{}{H_e, \ell_e, \ell_c, @_3 \Downarrow \mathit{err}}\ \textsc{red-app-3-no-ret}$$

$$\dfrac{H_e, \ell_e, \ell_c, e \Downarrow r \qquad r, \mathit{return}_1 \cdot \Downarrow r'}{H_e, \ell_e, \ell_c, \mathit{return}\, e \Downarrow r'}\ \textsc{red-return}(e)$$

$$\dfrac{}{(H_e, \ell_e, v), \mathit{return}_1 \cdot \Downarrow \mathit{ret}\,(H_e, \ell_e, v)}\ \textsc{red-return-1}$$

Figure 4.17: Rules added to manipulate functions

analysis more complex, this program does not return the result of the computation (it always returns the value 0) but stores it in a global variable r.

$$\begin{aligned}&\mathsf{prod} := (\lambda \mathsf{n}.\mathit{if}\ (\mathsf{n} > 0)\ (\mathsf{prod}\,(\mathsf{n} + (-1)); \mathsf{r} := \mathsf{r} + \mathsf{b})\ (\mathsf{r} := 0); \mathit{return}\, 0);\\ &\mathsf{a} := 6; \mathsf{b} := 7; \mathsf{z} := \mathsf{prod}\,(\mathsf{a})\end{aligned}$$

Running the freshly extracted analyser using the new abstract domain in an empty initial environment provides the following result, written in a readable form.

$$\left(\{\mathsf{r} \mapsto +,\ \mathsf{b} \mapsto +,\ \mathsf{a} \mapsto +,\ \mathsf{z} \mapsto 0\}, \overline{\mathit{err}}^{\sharp}\right)$$

This result is similar to what we have found in Section 4.6.2. The analyser detected again that no error can be returned by this program, and that the result if strictly positive. Notice that this is not trivial for a human in this new setting given how the analysed program has been written: at each step, we read the variable r, which is only initialised when $\mathsf{n} \leqslant 0$ by $\mathsf{r} := 0$. For this program to be valid, it is thus important to execute the recursive call *before* accessing r. Overall, very little changes were made to the definition and proofs to get a working certified analyser: the approach is indeed scalable—hopefully to JSCert.

4.7.2 Conclusion

This chapter described a framework to minimise the proof effort required to build abstract semantics and certified analyses. This framework is parametric in several aspects. First, it is parametric in the analysed language, which must be defined as a pretty-big-step semantics based on transfer functions (see Section 4.3). Second, it is parametric in the abstract domains, which must be defined, along with the corresponding abstract transfer functions. Once these functions are shown to soundly abstract the concrete transfer functions (at a local scope), a sound-by-construction abstract semantics is automatically defined. From this abstract semantics, an analysis can be developed. The framework provides several generic analysers which do not need to be adapted and can be run on any semantics, but ad-hoc analysers can also be defined for specific situations to which these generic analysers are not adapted. As we have seen in the previous section, this framework enables quick extensions of the semantics with little effort.

The pretty-big-step semantics built in this framework has more constraints than the pretty-big-step style presented in Section 2.5.2.1: the syntactic and the semantic aspect of rules have to be clearly separated, the transfer functions have to explicitly appear, as well as the side-conditions. Another difference is that this framework is less typed that JSCert: apart from the presentation of Section 4.5, the types of semantic contexts and results are independent from the evaluated term; whilst JSCert separates the expression reduction $\Downarrow_e$ from the statement reduction $\Downarrow_s$ by given them separate types. In this respect, the presented framework is similar to the reduction predicate $\Downarrow_i$ of JSCert. Updating the JSCert specification to this framework should probably be long, but straightforward.

As a conclusion, we now consider which abstract rules presented in Section 4.2 can be expressed in this framework. Rule IF-ABS-CORRECTED applies both premises of an *if*-construct in parallel; as we have seen in Figure 4.8b, this kind of rule derives automatically from the definition of the abstract immediate consequence $\mathcal{F}^\sharp$. Rule WHILE-ABS-FIXED-POINT enables to analyse *while*-constructs from an invariant of the state. In this framework, this is caught by the coinductive nature of the abstract derivations presented Section 4.4.2.2, or by Park's principle as presented in Section 4.6. Figure 4.9 shows such an infinite derivation defined using an invariant. Note that coinduction enables much more complex abstract derivations in which the abstract state is changed at each step—in other words, the Park set proven to be coherent can be infinite—: this framework does not explicitly require an invariant to hold, coinduction being much more permissive. We have seen in Section 4.6.1 that to accept Rule ABS-TOP, we need a rule similar to Rule ABS-WEAKEN. This last rule belongs to another kinds of rules called *structural* rules, treated in the next chapter. Structural rules also include Rule GLUE-TRACE-PARTITIONING, which enables to derive equivalents of Rules IF-ABS-REFINED and WHILE-ABS-PRECISE-FIXED-POINT.

5 Structural Rules

Tu vas te faire mal inutilement… avec cette colle-là, il n'y a qu'un système : l'eau bouillante !

Gaston Lagaffe, by André Franquin [FD70]

Chapter 4 presented a basic framework to build certified abstract semantics without having to deal with complex proofs. We have seen in Section 4.7.1 how this framework can be extended to consider new rules, without having to prove any complex new invariant. But this last chapter concluded in Section 4.7.2 that this framework is incomplete as-is.

This chapter explains how we can add structural rules—also called *glue* rules—into this formalism, in particular both Rules GLUE-WEAKEN and GLUE-TRACE-PARTITIONING which we needed in the previous chapter. We first explain why structural rules are different from the other rules—which we call *computational*—, then extend the formalism to catch these additional rules. Section 5.3 details the COQ proof of soundness in this extended formalism. This chapter uses the language defined in Figures 3.4 and 4.2 for most of its examples.

5.1 Examples of Structural Rules

As recalled at the end of Section 3.3, the goal of an abstract semantics is not to be precise, but to specify what are *acceptable* semantic triples. In other words, an abstract semantics specifies which abstract results can be considered correct. In a second step, once the abstract semantics is defined, we can define analysers, whose goal is to be computable, and if possible, precise. In most cases, a precise abstract semantics (in the sense that it only accepts precise results) is useless, as it does not relate to any not-so-precise analyser. In order for an abstract semantics to accept as many results as possible, we need somespecial rules not related to any concrete rule. We call these rules *structural rules*. This section presents two of such rules and how they can be used in practical analyses.

5.1.1 Approximations

Analysers make compromises about preciseness to ensure computability. Typically, when analysing a loop such as $while_1\ (\mathsf{x} > 0)\ (\mathsf{x} := \mathsf{x} + (-1))$ (as in Figure 4.9), analysers use techniques such as widening and narrowing [CC77a] to find loop invariants faster. To enable the use of such techniques in certified analysers, we need Rule GLUE-WEAKEN defined in Figure 5.1a. This rule states that to prove an abstract semantic triple $\sigma^\sharp, t \Downarrow r^\sharp$, the

$$\frac{\sigma^\sharp \sqsubseteq \sigma'^\sharp \qquad \sigma'^\sharp, t \Downarrow r'^\sharp \qquad r'^\sharp \sqsubseteq r^\sharp}{\sigma^\sharp, t \Downarrow r^\sharp}\ \textsc{glue-weaken}$$

(a) Definition

$$\cfrac{\cfrac{\cfrac{\vdots}{\sigma^\sharp, t \Downarrow r^\sharp}\ \textsc{glue-weaken}}{\sigma^\sharp, t \Downarrow r^\sharp}\ \textsc{glue-weaken}}{\sigma^\sharp, t \Downarrow r^\sharp}\ \textsc{glue-weaken}$$

(b) Unsound infinite derivation

Figure 5.1: Rule GLUE-WEAKEN

semantic context $\sigma^\sharp$ can be replaced by a greater one $\sigma'^\sharp$ in the abstract poset. This is sound as the soundness (as stated by Theorem 4.1) is about not missing any concrete behaviours, and the constraints of Section 3.2.4 enforce the concretisation function γ (and thus, the represented concrete behaviours) to be compatible with the order: $\gamma\left(\sigma^\sharp\right) \subseteq \gamma\left(\sigma'^\sharp\right)$. Rule GLUE-WEAKEN then continues the computation up to a result $r'^\sharp$, which can also be replaced by a greater element $r^\sharp$. This rule does not correspond to any concrete rule—its soundness is based on how derivations are built—and thus constitutes a structural rule.

Rule GLUE-WEAKEN is thus locally sound. It is also non-deterministic: there may be several instances of $\sigma'^\sharp$ greater than $\sigma^\sharp$ in the abstract poset. As mentioned in Section 3.1, the non-determinism of the abstract semantics is not an issue—on the contrary: it enables analysers to be flexible about the semantic triples accepted by the abstract semantics. In particular, an analyser can use any heuristic to chose which abstract rule to apply.

Rule GLUE-WEAKEN updates the result given by its subderivation. It thus does not follow the pretty-big-step style (see Section 4.3.1). Also, Rule GLUE-WEAKEN does not correspond to any concrete rule. This makes Rule GLUE-WEAKEN a rule of a different kind than the abstract rules presented in Section 4.4.1.1. In particular the soundness theorem (Theorem 4.1) does not apply. Consider Figure 5.1b, which continuously applies Rule GLUE-WEAKEN starting from any semantic triple $\left(\sigma^\sharp, t, r^\sharp\right)$. As the poset order $\sqsubseteq$ is reflexive, changing neither the semantic context nor the result is a valid choice for Rule GLUE-WEAKEN. The resulting derivation is infinite, and thus coinductively defined—although the semantic triple is not constrained. This derivation thus accepts invalid semantic triples: it is unsound. We need a new mechanism to deal with this new kind of rule.

5.1.2 Trace Partitioning

Figure 4.8 shows two derivations of the program *if* $(\mathsf{x} > 0)$ $(\mathsf{r} := \mathsf{x})$ $(\mathsf{r} := 18)$, one where x has the abstract value $+$ in the abstract environment, and one in which it has the abstract value $\top_{\mathbb{Z}}$. One would expect x to be positive in the "positive" branch, but this is not what happens in practise, as Rule RED-IF-1-POS$(\mathsf{r} := \mathsf{x}, \mathsf{r} := 18)$ has been abstracted in a very simple way which does not filter the semantic context to fit the side-condition. A more complex way would be to make use of the $cond_{\mathfrak{r}}(\sigma)$ premise of Criterion 4.5, repeated below: this criterion enables to only abstract the semantic contexts which fire the concrete side-condition—in this case, to select and only abstract positive values of x.

$$\forall \sigma^\sharp, \sigma \in \gamma\left(\sigma^\sharp\right). \; cond_{\mathfrak{r}}(\sigma) \rightarrow \\ up(\sigma) \text{ and } up^\sharp\left(\sigma^\sharp\right) \text{ defined} \rightarrow up(\sigma) \in \gamma\left(up^\sharp\left(\sigma^\sharp\right)\right) \quad \text{(4.5 repeated)}$$

To update the semantic context in Rule RED-IF-1-POS(s_1, s_2), we need to add information in the abstract domain. First, we need to know the *if*-condition after entering a branch: this condition has been removed at this stage of the computation (see Rule RED-IF(e, s_1, s_2) of Figure 3.4b). This is due to the way computations are performed in pretty-big-step (see Section 2.5.2.1), but the problem would also arise in other semantic styles with more complex examples. The abstract Rules RED-IF(e, s_1, s_2) and RED-IF-1-POS(s_1, s_2) below show how we can transport the *if*-condition e into the abstract domain.

$$\text{RED-IF}(e, s_1, s_2)\quad \frac{E^\sharp, e \Downarrow r^\sharp \qquad e, E^\sharp, r^\sharp, if_1\; s_1\; s_2 \Downarrow r'^\sharp}{E^\sharp, if\; (e > 0)\; s_1\; s_2 \Downarrow r'^\sharp} \qquad \text{RED-IF-1-POS}(s_1, s_2)\quad \frac{update_e\left(E^\sharp\right), s_1 \Downarrow r^\sharp}{e, E^\sharp, v, if_1\; s_1\; s_2 \Downarrow r^\sharp}\; v > 0$$

The carried expression e is ignored when defining the concretisation: we have $\gamma\left(\left(e, E^\sharp\right)\right) = \gamma\left(E^\sharp\right)$. This expression carries non-local information about the derivation, but does not influence the represented abstract semantic contexts. The expression e is thus transferred to Rule RED-IF-1-POS(s_1, s_2), and enables us to define a function $update_e$ as follows:

$$\begin{aligned} update_e\left(E^\sharp\right) &= E^\sharp\left[\mathsf{x} \leftarrow E^\sharp[\mathsf{x}] \sqcap +\right] && \text{if } e = \mathsf{x} \\ update_e\left(E^\sharp\right) &= E^\sharp && \text{otherwise} \end{aligned}$$

The function $update_e$ corresponds to what is called a backward analysis of expression in abstract interpretation [Jou16, Chapter 6]. Not all expressions can be easily covered: if instead of considering the rules of Figures 3.4 and 4.2, we consider those of Figures 4.15 and 4.17, which include function calls potentially updating the environment, then defining such a backward inference is as difficult as analysing a program. As a consequence, we only define an action for the function *update* on trivial expressions: when a variable x is given, we select the positive part of its value (we know that it can not be negative or undefined in this branch). Continuing into this direction leads to much more precise rules similar to Rules IF-ABS-REFINED and WHILE-ABS-PRECISE-FIXED-POINT of Figure 4.4.

Figure 4.8a shows that the formalism is able to deal with precise semantic contexts. Trace partitioning [MR05; RM07] presents an alternative: instead of defining complex abstract rules, we can separately consider precise values. For instance if a variable is associated the value $\top \in Store^\sharp$, we can separate the cases where its abstract value is $-_0$, $+$, and $undef^\sharp$. More generally, the semantic context $\sigma^\sharp$ of a semantic triple can be split into the contexts $\sigma_1^\sharp, \ldots, \sigma_n^\sharp$ if $\gamma\left(\sigma^\sharp\right) \subseteq \gamma\left(\sigma_1^\sharp\right) \cup \ldots \cup \gamma\left(\sigma_n^\sharp\right)$—in other words, if no concrete behaviour is missed. In practise, we try to minimise $\gamma\left(\sigma_1^\sharp\right) \cup \ldots \cup \gamma\left(\sigma_n^\sharp\right)$—ideally making it equal to $\gamma\left(\sigma^\sharp\right)$—to avoid introducing new concrete states. Figure 5.2 pictures this constraint. Rule GLUE-TRACE-PARTITIONING is shown below. It can have more than two premises: as for Rule GLUE-WEAKEN, it does not respect the pretty-big-step format.

$$\text{GLUE-TRACE-PARTITIONING}\quad \frac{\sigma_1^\sharp, t \Downarrow r^\sharp \qquad \ldots \qquad \sigma_n^\sharp, t \Downarrow r^\sharp}{\sigma^\sharp, t \Downarrow r^\sharp} \quad \gamma\left(\sigma^\sharp\right) \subseteq \gamma\left(\sigma_1^\sharp\right) \cup \ldots \cup \gamma\left(\sigma_n^\sharp\right)$$

We can now analyse the program $\mathit{if}\ (\mathsf{x} > 0)\ (\mathsf{r} := \mathsf{x})\ (\mathsf{r} := 18)$. Figure 5.3 shows a derivation based on Rule GLUE-TRACE-PARTITIONING. It starts by splitting the abstract environment $\{\mathsf{x} \mapsto \top_{\mathbb{Z}}\}$ in the two environments $\{\mathsf{x} \mapsto +\}$ and $\{\mathsf{x} \mapsto -_0\}$. We indeed have $\gamma\left(\{\mathsf{x} \mapsto \top_{\mathbb{Z}}\}\right) = \gamma\left(\{\mathsf{x} \mapsto +\}\right) \cup \gamma\left(\{\mathsf{x} \mapsto -_0\}\right)$. In both cases, Rule RED-VAR(x) now provides precise results in accordance to their respective heap: the result of the if-condition is now related with the value of x in the heap. This enables us to precisely continue the construction of the derivation and to get the precise expected result $\{\mathsf{x} \mapsto +\}$ in both cases.

How and where to efficiently split depend on the expression from which we want to extract information, as well as on the domains used in the analysis. This can involve arbitrarily complex heuristics in analysers. In general, it is a good practise to split the state before branching rules such as Rules RED-IF(e, s_1, s_2) and RED-WHILE(e, s). In this dissertation, we do not consider how these rules can be applied. We showed that structural rules are necessary to build some derivations: both Figures 4.8b and 5.3 use structural rules. We do not provide any methods about where and how to use such rules. This thesis indeed focusses on building certified abstract semantics for large semantics (and JavaScript in particular), but not on how to implement analysers from these semantics.

5.2 The Immediate Consequence Operator

In the previous section, we have seen some examples of structural rules. We have also seen that these rules do not follow the restrictions of pretty-big-step, and in particular that Theorem 4.1 does not apply on them. This section explains how we can nevertheless update the formalism to take such rules into account.

5.2.1 Ensuring Productivity of Computational Rules

As mentioned in Sections 3.3 and 4.4.3, the soundness of our formalism is based on the coverage of concrete derivations by abstract derivations starting from related semantic contexts and terms. Structural rules have by definition no counterpart in concrete derivations. The problem with the derivations of Figure 5.1b is that these structural rules are applied infinitely, without any computational rule in between.

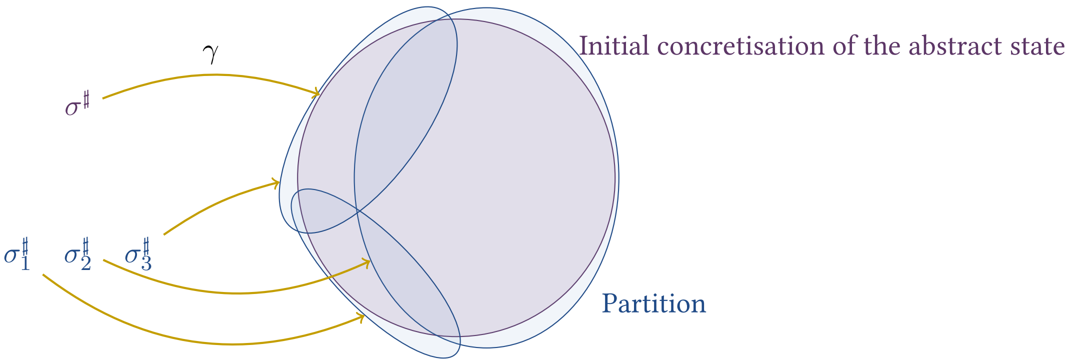


Figure 5.2: A picturisation of a trace partitioning

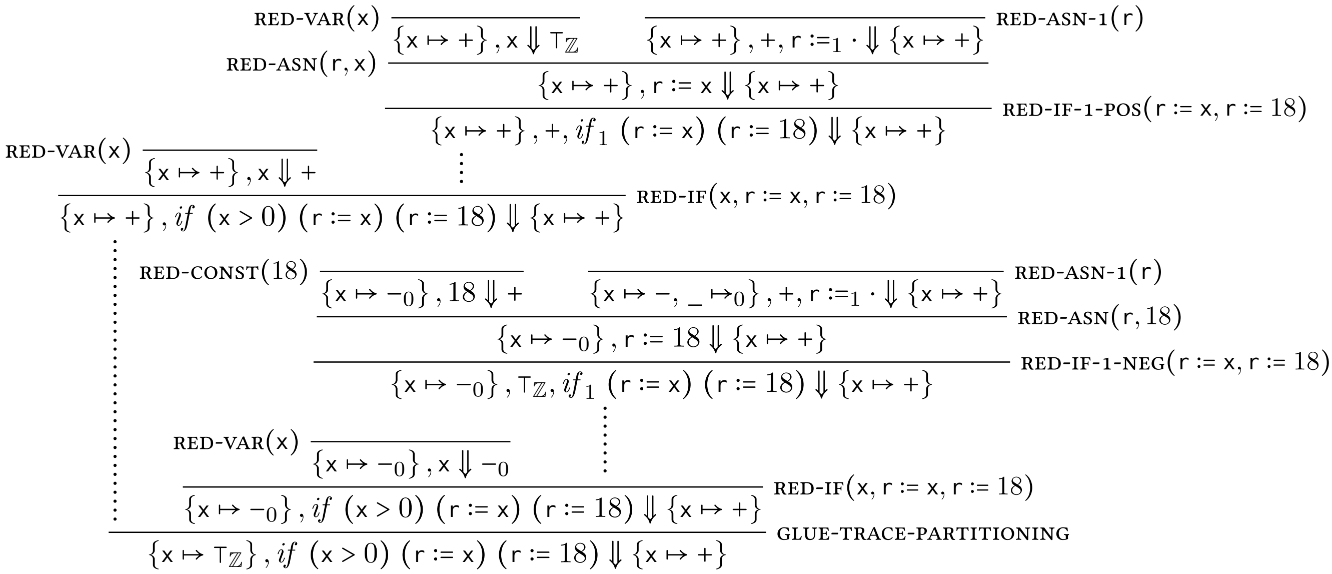


Figure 5.3: A derivation using trace partitioning

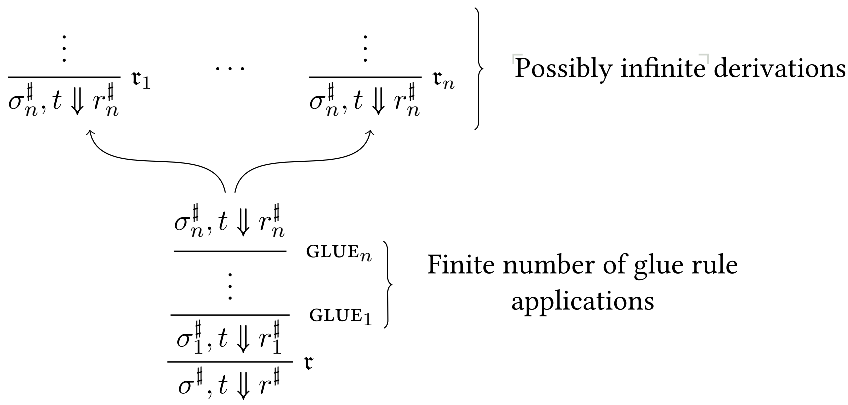


Figure 5.4: Illustration of an infinite abstract derivation with glue

Structural and computational rules have to be dealt differently in the immediate consequence $\mathcal{F}^\sharp$ operation. In particular, we have to make sure that the computational part of an abstract derivation is *productive*, that is, we can infer which are the next applied computational rules at each step of a derivation. In the case of the derivation of Figure 5.1b, it is not, as there is no computational rule in this derivation. To this end, we force structural rules to be only applied a finite amount of time (that is, inductively) between each computational rule, as shown in Figure 5.4. Structural rules are alternatively named *glue* rules because they apply between abstract rules without changing the derivation structure.

In our formalisation, structural rules are applied after the rule separation described in Section 4.4.2.1. These rules are given in the form of a predicate $glue : \mathcal{P}\left(st \times res\right) \to st \to res \to Prop$. The predicate $glue\left(\left(\sigma_i^\sharp, r_i^\sharp\right), \sigma^\sharp, r^\sharp\right)$ represents the following glue rule.

$$\frac{\forall i, \left(\sigma_i^\sharp, t \Downarrow r_i^\sharp\right)}{\sigma^\sharp, t \Downarrow r^\sharp}$$

For instance, Rules GLUE-WEAKEN and GLUE-TRACE-PARTITIONING are respectively associated with the predicates $glue_{\text{WEAKEN}}$, and $glue_{\text{TRACE-PARTITIONG}}$ defined below. Defining rules with predicates is generic and suits many formalisms. For instance, the form of Rules GLUE-FRAME-$\boxed{\star}$ and GLUE-FRAME-$\boxed{\odot}$ of Figure 6.9 is not common in abstract interpretation.

$$glue_{\text{WEAKEN}}\left(\left\{\sigma'^\sharp, r'^\sharp\right\}, \sigma^\sharp, r^\sharp\right) \iff \sigma^\sharp \sqsubseteq \sigma'^\sharp \wedge r'^\sharp \sqsubseteq r^\sharp \tag{5.1}$$

$$glue_{\text{TRACE-PARTITIONG}}\left(\left\{\sigma_i^\sharp, r^\sharp\right\}, \sigma^\sharp, r^\sharp\right) \iff \gamma\left(\sigma^\sharp\right) \subseteq \bigcup_i \gamma\left(\sigma_i^\sharp\right) \tag{5.2}$$

We now update the definition of the immediate consequence $\mathcal{F}^\sharp$ from Section 4.4.2.1 as follows. It now consists of three steps (instead of two). First the rules are filtered to get those which applies. Second, the glue is applied. Third, transfer functions are computed. As for the old abstract immediate consequence $\mathcal{F}^\sharp$, the abstract semantics can now be defined by iterating $\mathcal{F}^\sharp$ from an empty seed.

$$\mathcal{F}^\sharp\left(\Downarrow_0\right) = \left\{ \left(\sigma^\sharp, t, r^\sharp\right) \;\middle|\; \begin{array}{l} \forall \mathfrak{r}.\, t = \mathfrak{l}_\mathfrak{r} \to cond_\mathfrak{r}^\sharp\left(\sigma^\sharp\right) \\ \quad \to \exists \left(\sigma^\sharp\right)_i, \left(r^\sharp\right)_i .\ glue\left(\left\{\left(\sigma_i^\sharp, r_i^\sharp\right)\right\}, \sigma^\sharp, r^\sharp\right) \\ \quad\quad \wedge \forall i.\ \left(\sigma_i^\sharp, t, r_i^\sharp\right) \in apply_\mathfrak{r}\left(\Downarrow_0\right) \end{array} \right\}$$

Up to now, we have assumed a predicate *glue*, giving some instances. We now provide the constraint which we impose on such predicate to provide a sound abstract semantics.

5.2.2 Correctness Criterion

As we have seen in Section 5.1, the glue rules are meant to catch global invariants—or at least, not as local as what transfer functions catch. The criterion which we require on the glue rules is based on their ability to rewrite concrete derivations to make them match its results. The soundness of the examples presented in Section 5.1 relies on the fact that concrete derivations can only be in some specific forms, which these rules take into account. We thus need a more complex example to show these rewritings in action.

Section 6.4.1 introduces the glue Rules GLUE-FRAME-⓪ and GLUE-FRAME-⊛, whose soundness crucially depends on such rewritings. We now present the essence of these rules. Consider the language defined in Section 4.7.1. This language features functions calls, and in particular, a heap of allocated environments. The rules of this language (see Figure 4.17) have been carefully defined so that environments in the environment heap H_e are never modified: every time we need to write in an environment, we allocate a new updated environment in the environment heap. Two of such concrete rules are shown below.

$$\textsc{red-asn-1}(\mathsf{x})\quad \frac{\ell'_e = \mathit{fresh}\left(H_e\right) \qquad E = H_e\left[\ell_e\right]}{\ell_c, \left(H_e, \ell_e, v\right), \mathsf{x} :=_1 \cdot \Downarrow H_e\left[\ell'_e \leftarrow E\left[\mathsf{x} \leftarrow v\right]\right], \ell'_e, \ell_c} \quad \mathsf{x} \notin \mathit{dom}\left(H_e\left[\ell_c\right]\right)$$

$$\textsc{red-asn-1-local}(\mathsf{x})\quad \frac{\ell'_c = \mathit{fresh}\left(H_e\right) \qquad C = H_e\left[\ell_c\right]}{\ell_c, \left(H_e, \ell_e, v\right), \mathsf{x} :=_1 \cdot \Downarrow H_e\left[\ell'_c \leftarrow C\left[\mathsf{x} \leftarrow v\right]\right], \ell_e, \ell'_c} \quad \mathsf{x} \in \mathit{dom}\left(C\right)$$

In this setting, the environment heap H_e keeps increasing during the execution of a program. Reusing previously computed semantic triples as in Section 4.6.2 appears to be a complex task, as the environment heap never matches. To this end, we introduce a partial operation $\uplus$ defined over semantic contexts and results. For each abstract environment heap $H_e^\sharp$, this operation completes the environment heap of the given semantic context $\sigma^\sharp$ or result $r^\sharp$ with the environments of $H_e^\sharp$. For instance[1], if $\sigma^\sharp = \left(H_e^\sharp, \eta_e, \eta_c\right)$, then $\sigma^\sharp \uplus H'^\sharp_e = \left(H_e^\sharp \uplus H'^\sharp_e, \eta_e, \eta_c\right)$. We do not detail how the operation $\uplus$ is defined. Section 6.4.1 provides a similar operation. Consider now the glue Rule FRAME-ENV below.

$$\textsc{frame-env}\quad \frac{\sigma^\sharp, t \Downarrow r^\sharp}{\sigma^\sharp \uplus H_e^\sharp, t \Downarrow r^\sharp \uplus H_e^\sharp}$$

This rule is sound in the sense that the statement of the soundness theorem 4.1 holds in the presence of this glue. Indeed, if we have produced an abstract semantic triple $\sigma^\sharp \uplus H_e^\sharp, t \Downarrow^\sharp r^\sharp \uplus H_e^\sharp$ using Rule FRAME-ENV, then we have by hypothesis succeedingly built a sound abstract derivation for the abstract semantic triple $\sigma^\sharp, t \Downarrow^\sharp r^\sharp$. We now consider a concrete derivation of conclusion $\sigma, t \Downarrow r$ with $\sigma \in \gamma\left(\sigma^\sharp \uplus H_e^\sharp\right)$. We can prove that we can decompose the concrete semantic context σ in a semantic context σ_0 and a contextual environment heap $H_{e,0}$ such that $\sigma = \sigma_0 \bullet H_{e,0}$, where $\bullet$ is the concrete equivalent of the $\uplus$ operator. The $\bullet$ operator is defined similarly to the $\uplus$ operator: if $\sigma = \left(H_e, \ell_e, \ell_c\right)$, then $\sigma \bullet H'_e = \left(H_e \uplus H'_e, \ell_e, \ell_c\right)$.

[1] We are here using the abstract domains of Section 4.7.1, but the precise details about these domains are not needed to understand how Rule FRAME-ENV works.

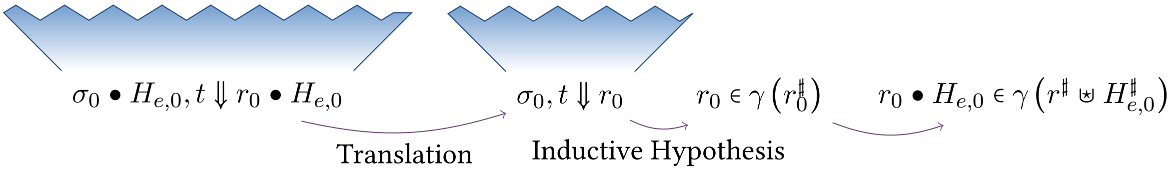


Figure 5.5: Structure of the proof that Rule FRAME-ENV is sound

```
Definition correct_up_to_depth k asigma ar :=
  forall n sigma r (A : apply n sigma r),
    gst asigma sigma →
    cond n sigma →
    apply_depth A < k →
    gres ar r.

Definition glue_correct := forall P asigma ar k,
  glue P asigma ar →
  (forall asigma' ar',
    P asigma' ar' →
    correct_up_to_depth k asigma' ar') →
  correct_up_to_depth k asigma ar.
```

Program 5.1: Coq definition of the correctness of glue rules

All the rules of our concrete language have been defined such that adding concrete (unreachable) environments to the environment heap H_e does not change neither the applicable rules, nor the results—apart from the fact that the added environments are conserved in the result. Similarly, removing environments from the environment heap H_e will either have no effect on a derivation, or will prevent the derivation to be built if we needed the removed environments: previously applicable rules will no longer apply and the derivation will be stuck. In particular, the result r of the above concrete derivation of conclusion $\sigma, t \Downarrow r$ with $\sigma = \sigma_0 \bullet H_{e,0}$ is of the form $r = r_0 \bullet H_{e,0}$. We now rewrite this concrete derivation to remove the environments from the context $H_{e,0}$ in all its intermediate states. This process may fail somewhere in the derivation because a rule needs the environments present in $H_{e,0}$ and no longer applies. This case is however not possible because we have successfully derived an abstract version of this derivation: in this abstract domain, abstract environments are particularly close to concrete environments. In particular, abstract derivations would fail if a concrete derivation does. This statement relies on the fact that different abstract locations η represent different concrete locations ℓ_e—Section 6.4.5 elaborates on this matter. We thus get a derivation of the form $\sigma_0, t \Downarrow r_0$. The structure of this derivation—that is, all the applied rules—is identical to the original derivation. By recursion[2], we get $r_0 \in \gamma\left(r_0^\sharp\right)$. Given the way the operators $\uplus$ and $\bullet$ have been defined, this yields $r_0 \bullet H_{e,0} \in \gamma\left(r_0^\sharp \uplus H_{e,0}^\sharp\right)$. Figure 5.5 sums up this proof.

[2] This paragraph only aims at giving an intuition of why Rule FRAME-ENV is sound. In particular, we do not detail this recursion here, which is in fact not so trivial. See Section 5.3 for a detailed proof.

```
Inductive glue_iter : name → (ast → ares → Prop) → ast → ares → Prop :=
  | glue_iter_refl : forall n (P : ast → ares → Prop) asigma ar,
    P asigma ar → glue_iter n P asigma ar
  | glue_iter_cons : forall n (P P' : ast → ares → Prop) asigma3 ar3,
    (forall asigma2 ar2, P' asigma2 ar2 → glue_iter n P asigma2 ar2) →
    glue n P' asigma3 ar3 →
    glue_iter n P asigma3 ar3.
```

Program 5.2: Coq definition of the iterating glue predicate

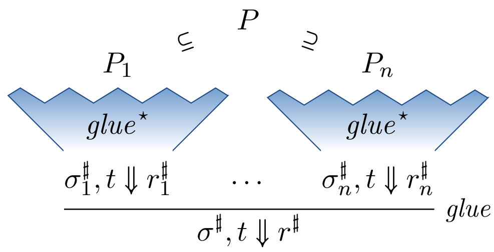


Figure 5.6: Intuition behind the definition of the iterating glue predicate $glue^\star$

The important aspect of these concrete derivation rewritings is that they do not change the depth of concrete derivations. To this end, we introduce an intermediary definition for the correctness of glue rules, associated with the depth k of the rewritten derivations. We define the k correctness as follows. A semantic triple $\sigma^\sharp, t \Downarrow^\sharp r^\sharp$ is k correct, k being a number, if for any concrete derivation of depth less than k with conclusion $\sigma, t \Downarrow r$, then $\sigma \in \gamma\left(\sigma^\sharp\right)$ implies $r \in \gamma\left(r^\sharp\right)$. We can now state the criterion for glue rules. The glue predicate $glue$ is correct if for each of its instances and for all k, the k correctness of its premises implies the k correctness of its conclusion:

$$\forall \left(\sigma_i^\sharp\right), \left(r_i^\sharp\right).\ glue\left(\left\{\left(\sigma_i^\sharp, r_i^\sharp\right)\right\}, \sigma^\sharp, r^\sharp\right) \implies \left(\forall i.\ \sigma_i^\sharp, t \Downarrow^\sharp r_i^\sharp \text{ is } k \text{ correct}\right) \implies \sigma^\sharp, t \Downarrow^\sharp r^\sharp \text{ is } k \text{ correct} \tag{5.3}$$

Lines 1 to 6 of Program 5.1 shows the Coq definition of k correctness. In Coq, there are several predicates to define derivations, as shown in Program 4.2: `eval` corresponds to $\Downarrow$, and `apply` corresponds to the step in which we already chose the applied rule. The type `apply n`—`n` being a rule name—represents a concrete derivation whose root rule is `n`. The predicates `gst` and `gres` correspond to the concretisation functions of the abstract domain. The Coq equivalent of Criterion 5.3 is shown Lines 8 to 13 of Program 5.1.

Note that in the definition of the immediate consequence $\mathcal{F}^\sharp$, we force the glue to be applied at each step. We furthermore suppose to only have one glue predicate $glue$. The next section explores how to circumvent this constraints.

5.2.3 Lifting to Several Rules

We have claimed in Figure 5.4 that we inductively (as opposed to coinductively) apply glue rules at each step of an abstract derivation, but Section 5.2.1 updated the immediate consequence to consider exactly one glue rule at each step. This section introduces two glue rules: one to mix several glue predicates and one to iterate on a glue predicate. Once these two rules are defined, we can consider a finite number[3] of glue predicates $(glue)_i$ and build the glue $(glue_1 + \ldots + glue_n)^\star$—notations being as can be expected. This is equivalent to considering a finite number of glue rules and applying them inductively.

We start by the sum glue. We assume two glue predicates $glue_1$ and $glue_2$, both respecting Criterion 5.3. The sum glue $glue_1 + glue_2$ is defined as the set union of both glues:

$$(glue_1 + glue_2)\left(P, \sigma^\sharp, r^\sharp\right) \iff glue_1\left(P, \sigma^\sharp, r^\sharp\right) \vee glue_2\left(P, \sigma^\sharp, r^\sharp\right)$$

To use this glue, we have to prove it correct—that is, it respects Criterion 5.3—assuming both initial glue predicates $glue_1$ and $glue_2$ are correct. We introduce the first hypothesis of Criterion 5.3: there exists a set P (standing for "premises"), an abstract semantic context $\sigma^\sharp$, and an abstract result $r^\sharp$ such that $(glue_1 + glue_2)\left(P, \sigma^\sharp, r^\sharp\right)$. By definition of the sum glue, we have either $glue_1\left(P, \sigma^\sharp, r^\sharp\right)$ or $glue_2\left(P, \sigma^\sharp, r^\sharp\right)$. We conclude by applying the correctness of the corresponding initial glue.

We now define the iterating glue from a glue predicate $glue$ respecting Criterion 5.3. The predicate $glue^\star$ is defined as the smallest predicate respecting the following conditions.

$$\forall P, \left(\sigma^\sharp, r^\sharp\right) \in P.\ glue^\star\left(P, \sigma^\sharp, r^\sharp\right) \tag{5.4}$$

$$\begin{aligned}&\forall P, P', \sigma^\sharp, r^\sharp.\ glue\left(P', \sigma^\sharp, r^\sharp\right) \implies \\ &\qquad \left(\forall \left(\sigma'^\sharp, r'^\sharp\right) \in P'.\ glue^\star\left(P, \sigma'^\sharp, r'^\sharp\right)\right) \implies glue^\star\left(P, \sigma^\sharp, r^\sharp\right)\end{aligned} \tag{5.5}$$

This translates into Coq as an inductive definition, as shown in Program 5.2. Sets of pairs are represented in Coq by predicates with two arguments. Intuitively Condition 5.5 states that the predicate $glue^\star$ can run $glue$ once, then iterate on its premises. The newly constructed premises can then all be merged into the set P using Condition 5.4. Figure 5.6 pictures this intuition. The definition of $glue^\star$ differs from this intuition in one major aspect: it assumes that each set P_1 to P_n is equal to P.

To enable each set of premises P_i to be equal to P, we need the predicate $glue$ to accept weakenings on its premises, accepting more hypotheses. Given a glue predicate $glue$ respecting Criterion 5.3, we define the closure predicate $glue^c$ as follows.

$$glue^c\left(P, \sigma^\sharp, r^\sharp\right) \iff \exists P_0 \subseteq P.\ glue\left(P_0, \sigma^\sharp, r^\sharp\right)$$

[3] The sum glue only mixes two glue predicates, but it could easily be extended to an infinite number. We only consider a finite number of glue predicates (which are really rule *schemes*) for readability purposes. The only important matter is to apply a finite amount of glue rules between any two computational rules.

We thus accept more sets P, as soon as they include the original set of premises P_0. This glue predicate clearly respects Criterion 5.3, as it restricts its application to cases where more k correct derivations are given. The glue predicate $glue^c$ can be understood as the fact that if a rule is sound, so does any similar rule with additional premises.

It is thus possible, given any number of correct glue predicates $(glue)_i$, to assemble them into a single glue predicate: by summing all them into a single predicate $glue$ and taking its closure $glue^c$ (or the converse: these two operations commute), then iterating it. We have seen that these operations conserve the correctness of glue predicates. The resulting predicate $(glue_1^c + \ldots + glue_n^c)^\star$ enables to (finitely) apply each predicate $glue_i$ as many times as wanted between any two computational rules. This property justifies the rule notation of the glue. We now present the proof of soundness in this new setting.

5

5.3 Proof of Soundness

We update the soundness theorem (Theorem 4.1) to take into account the glue. In essence, its statement does not change: abstract derivations capture every concrete derivations. This is the soundness statement of Schmidt, as we have seen in Section 3.3. The translation of this theorem in Coq is shown in Program 5.3, in Lines 7 to 11.

Theorem 5.1 (Soundness). *Given the (local) soundness of each abstract rule and the correctness of the glue, if we have $\sigma \in \gamma\left(\sigma^\sharp\right)$ as well as concrete and abstract derivations of respective conclusions $\sigma, t \Downarrow r$ and $\sigma^\sharp, t \Downarrow^\sharp r^\sharp$, then $r \in \gamma\left(r^\sharp\right)$.*

We have sketched in Section 4.4.3 a proof of Theorem 4.1 in the case without glue. The proof has been performed by induction over the concrete derivation of conclusion $\sigma, t \Downarrow r$. Induction in Coq is limited to ensure termination: the inductive hypothesis should only be applied on strictly smaller derivations. This is unfortunately no longer the case here, as the correctness of the glue rules is based on rewriting the concrete derivation. But as we said in Section 5.2.2, glue rules preserve the depth of the rewritten derivations. The decreasing argument of this induction is thus chosen to be the depth k of the concrete derivation. Apart from this change, the proof follows the proof sketch of Section 4.4.3.

Program 5.3 presents the Coq proof of Theorem 5.1. Line 13 introduces two derivations: the concrete derivation `D`, and the abstract derivation `aD`. The term `G` is a proof that $\sigma \in \gamma\left(\sigma^\sharp\right)$. The semantic context $\sigma^\sharp$ is written `asigma` in Coq (and parts of the abstract world are generally prefixed by the letter `a`). Line 14 starts the induction on the depth `k` of the derivation. The `gen` tactic enables to generalise some variables on which the induction should not depend. The inductive hypothesis follows.

Hypothesis 5.2. *Given a number d, then for all term t, for all related concrete and abstract semantic context σ and $\sigma^\sharp$, for all concrete and abstract results r and $r^\sharp$, for all concrete derivation of depth d with $\sigma, t \Downarrow r$ as a conclusion, and for all abstract derivation with $\sigma^\sharp, t \Downarrow r^\sharp$ as a conclusion, the concrete and abstract results r and $r^\sharp$ are related.*

```
Hypothesis transfer_functions_sound : forall n,
  propagates (acond n) (cond n) (arule n) (rule n).
Hypothesis acond_sound : forall n asigma sigma,
  gst asigma sigma → cond n sigma → acond n asigma.
Hypothesis glue_is_correct : glue_correct glue.

Theorem soundness : forall t asigma ar,
  aeval asem glue t asigma ar →
  forall sigma r, gst asigma sigma →
    eval sem t sigma r →
    gres ar r.
Proof.
  introv aD G D.
  gen t asigma sigma r ar. induction (eval_depth D) as [|k].
   math.
   introv G I aD. destruct D as [E C A] eqn: ED. inverts aD as allBranches.
    forwards~ aA: allBranches E.
      apply* acond_sound.
    clear E. inverts aA as Gl aA. forwards~: glue_is_correct Gl A I.
    introv OK G0 C0 I0. forwards aA’: aA OK.
    forwards TrCn: transfer_functions_sound n.
    destruct A0 as
        [ E ax sigma0 r0
        | E t0 up sigma0 sigma1 r1 D0
        | E t1 t2 up next sigma0 sigma1 sigma2 r1 r2 D1 E4 D2];
      destruct aA’ as
        [ aE aax asigma0 ar0
        | aE at0 aup asigma0 asigma1 ar1 aD0
        | aE at1 at2 aup anext asigma0 asigma1 asigma2 ar1 ar2 aD1 aE4 aD2];
        inverts TrCn as TrC1 TrC2; rewrite aE in E; inverts E.
    (** Axiom **)
    apply* TrC1.
    (** Format 1 Rule **)
    applys~ IHk D0 aD0; [| math ]. apply* TrC1.
    (** Format 2 Rule **)
    applys~ IHk D2 aD2; [| math ].
     apply* TrC2.
     applys~ IHk D1 aD1; [| math ]. apply* TrC1.
Qed.
```

Program 5.3: Coq proof of Theorem 5.1

The depth function `eval_depth` has been defined to be non-zero on a derivation (an axiom application is considered to have the depth 1). This choice was made to avoid separating the axiom case from the other two cases. As a consequence, the base step of the induction is trivial, treated by the `math` tactic Line 15. Line 16 destructs the concrete and abstract derivations: the concrete derivation reveals a rule name $\mathfrak{r}$ such that $\mathfrak{l}_{\mathfrak{r}} = t$ (whose proof certificate is named `E` in Coq), that its side-condition applies (named `C`), as well as the derivation continuation `A`. The abstract derivation produces `aA`, a universally quantified derivation for all rules which apply. We instantiate it Line 17 to Rule $\mathfrak{r}$. The proof then divides in two goals: the abstract side-condition should apply (as the concrete does), and the continuation of the derivation is sound. The first goal is directly solved by Criterion 4.2.

Line 19 uses Criterion 5.3 to get a new concrete derivation. This new derivation is not a subterm of the initial concrete derivation, but it has at most the same depth: Coq accepts the use of the induction hypothesis 5.2 on this new concrete derivation. Lines 22 to 30 destructs the concrete and the abstract derivation; each is destructed into three goals (corresponding to the three kinds of rules in pretty-big-step, shown in Figure 4.5). The formats of the concrete and abstract rule applications should correspond: Line 30 removes the conflicting combinations to leave three goals in total out of the nine generated.

The three cases are then straightforward: the local soundness assures local propagation, and the premises are dealt by the inductive hypothesis. For instance, Line 34 solves the requirements for format 1 rules. We apply here the inductive hypothesis `IHk` with the concrete and abstract subderivations `D0` and `aD0`. These subderivations come from the destruction of Lines 22 to 30. Coq leaves two goals. The first goal is a proof that the depth of the new concrete derivation is indeed smaller than the current. This goal is automatically handled by the `math` tactic. The second goal is a proof that the premise of the inductive hypothesis holds: we have to show that $up(\sigma) \in \gamma\left(up^\sharp\left(\sigma^\sharp\right)\right)$. This is exactly given by the local soundness `TrCn` of the transfer function constructed Line 21. Overall, the Coq proof is very similar to the proof presented in Section 4.4.3.

5.4 Conclusion

In this chapter, we have extended the formalism of Chapter 4 by introducing structural rules, or glue rules. We have proven in Coq that the main theorem of the formalism—Theorem 5.1—still holds if Criterion 5.3 applies on the glue. These rules can be added to express new ways of reasoning, in particular non-local ones. The next chapter evaluates this formalism by introducing a rule from separation logic: the frame rule. This rule is indeed based on different assumptions and ways of reasoning than abstract interpretation, which tends to make mixing separation logic and abstract interpretation difficult.

6 Separation Logic and JavaScript

> *We recognize the fact that if different robots are subject to narrow definitions of one sort or another, there can only be measureless destruction.*
>
> Daneel Olivaw, by Isaac Asimov [Asi85]

In the previous chapters, we have presented methods to deal with JavaScript's complexity. Chapter 2 presented how to build a trustable concrete semantics of JavaScript, and Chapter 4 proposed to use this concrete semantics to guide the construction of sound-by-construction abstract semantics and analysers. The proposed abstract semantics is parametrised by its abstract domains. Up to now, we have only proposed very simple abstract domains, such as those of Section 4.1. This chapter aims at building abstract domains for the memory model of JavaScript, as presented in Section 1.2.3. The results presented in this chapter are not yet published at the time of this writing.

Separation logic [IO01; ORY01; Rey02; Rey08] aims at abstracting the heap. It has proven its abilities to provide strong and precise guarantees for JavaScript [GMS12]. Separation logic comes with a special structural rule called the frame rule, similar to Rule frame-env of Section 5.2.2. This is an opportunity to evaluate the formalism of Chapter 5. We use a simple variant of separation logic based on shape analysis [SRW98]. This variant aims at presenting different aspects of separation logic whilst being generic enough to be used in the analysis of interesting programs. This chapter is accompanied by a Coq formalisation [Bod16], whose structure is shown in Figure 6.1. It is divided into four steps: the pretty-big-step formalism (corresponding to Chapters 4 and 5), a concrete semantics, the definition of abstract domains, and the definition and local soundness proof of the abstract semantics. Each of these parts only depends on the previous parts. In particular, the formalism of Chapters 4 and 5 has not been changed to account for separation logic. This formalisation proved to be more ambitious than expected, and is unfortunately not yet finished. We do not consider this as a major issue: the goal of this chapter is to provide directions on how to apply the formalism of Chapters 4 and 5 in a more general setting. Separation logic is known to difficultly mix with abstract interpretation, and our formalisation provides a surprisingly deep insight on how to mix these two formalisms.

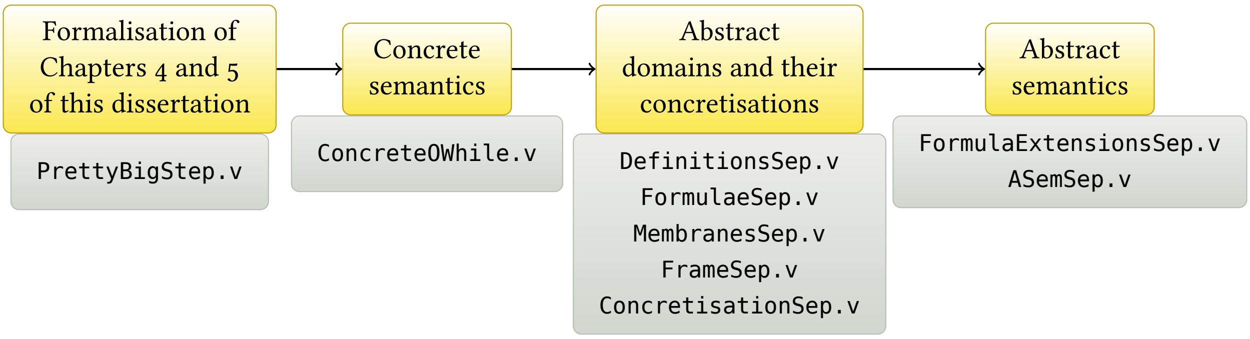


Figure 6.1: General structure of the Coq formalisation

Section 6.1 starts by extending the language of Chapter 4 to include a heap inspired from JavaScript's. Separation logic is introduced in Section 6.2. Section 6.3 presents how separation logic can be used in the context of the framework presented Chapter 4. The frame rule is put aside and treated in details in Section 6.4. Section 6.5 then extends this basic framework to include approximations.

We now introduce a notation used thorough this chapter. Given two sets A, B, and a function $f : A \to \mathcal{P}(B)$, we define the function $\dot{f} : \mathcal{P}(A) \to \mathcal{P}(A)$ as $\dot{f}(C) = \bigcup_{c \in C} f(c)$.

6.1 Language Extension: Adding a Heap

The heap is a critical part of JavaScript's semantics. The language which we consider is this chapter is an extension of the language defined in Section 4.7.1, a simple imperative language with functions as first class citizens. We extend its syntax and semantics to manipulate JavaScript-like dynamic objects. Objects o are represented as finite maps from fields $\mathtt{f} \in \mathit{Field}$ to values. The set Field is supposed to be infinite. We use a JavaScript-like notation for objects: the empty object is written $\{\}$, and an object mapping $\mathtt{f}$ to v_1 and $\mathtt{g}$ to v_2 is written $\{\mathtt{f} : v_1, \mathtt{g} : v_2\}$. Values can be either basic values $b^\sharp$ in $\mathbb{Z}$, closures $(\ell_c, \lambda \mathsf{x}.s)$, or locations $\ell \in \mathcal{L}$. New objects are allocated with the *alloc* expression, and the values of fields are obtained with the expression $e.\mathtt{f}$. The language includes constructs for writing to a field ($e_1.\mathtt{f} := e_2$), for deleting a field from an object (*delete* $e.\mathtt{f}$), and for testing the presence of a field ($\mathtt{f}$ *in* e). As usual, each of these new constructs comes with their extended terms. The syntax of our language is presented in Figure 6.2.

The program state is updated to carry an object heap H in addition to the environment heap H_e and the global and context environment locations ℓ_e and ℓ_c. Object heaps are finite maps from object locations $\ell \in \mathcal{L}$ to objects $o : \mathit{Field} \rightharpoonup_{\mathit{fin}} \mathit{Val}$. We do not define any *nil* or **`null`** additional value: in the examples of this chapter, linked lists will be ended by the 0 value. Expression results are either errors or a triple of an object heap H, an

$$
\begin{array}{llll}
e ::= c \in \mathbb{Z} & e_e ::= \cdot +_1 e & s \in stat ::= skip & s_e ::= \mathtt{x} :=_1 \cdot \\
\quad \mid \mathtt{x} \in Var & \quad \mid \cdot +_2 \cdot & \quad \mid s_1 ; s_2 & \quad \mid \cdot ;_1 s_2 \\
\quad \mid e_1 + e_2 & \quad \mid @_1 (e_2) & \quad \mid \mathtt{x} := e & \quad \mid if_1\ s_1\ s_2 \\
\quad \mid \lambda \mathtt{x}. s & \quad \mid @_2 & \quad \mid if\ (e > 0)\ s_1\ s_2 & \quad \mid while_1\ (e > 0)\ s \\
\quad \mid e_1 (e_2) & \quad \mid @_3 & \quad \mid while\ (e > 0)\ s & \quad \mid while_2\ (e > 0)\ s \\
\quad \mid alloc & \quad \mid .\mathtt{f} & \quad \mid return\ e & \quad \mid return_1 \cdot \\
\quad \mid e.\mathtt{f} & \quad \mid \mathtt{f}\ in_1 \cdot & \quad \mid e_1.\mathtt{f} := e_2 & \quad \mid .\mathtt{f} :=_1 e_2 \\
\quad \mid \mathtt{f}\ in\ e & & \quad \mid delete\ e.\mathtt{f} & \quad \mid .\mathtt{f} :=_2 \cdot \\
 & & & \quad \mid delete_1\ .\mathtt{f}
\end{array}
$$

Figure 6.2: Updating the language of Figure 4.14

environment heap H_e, an environment location ℓ_e, and a value. The semantic is defined such that each location present in an environment or returned by a value is a valid location, that is, the heap associates each referenced location to an object.

The additional concrete rules are presented in Figure 6.3. The full semantics is available in the companion website [Bod16]. Rules RED-FIELD-1($\mathtt{f}$), RED-IN-1-TRUE($\mathtt{f}$), and RED-IN-1-FALSE($\mathtt{f}$) check whether an object is present in the heap H at a given location ℓ, and whether this object possesses a field $\mathtt{f}$—we write this $\ell.\mathtt{f} \in dom^2(H)$. We allow to split heaps at the level of fields, writing $H_1 \uplus H_2$ (pronounced "field-union") when $dom^2(H_1) \cap dom^2(H_2) = \varnothing$ for the heap mapping each location $\ell \in dom(H_1) \cup dom(H_2)$ to the object $o_1 \uplus o_2$ (see the notations for maps at the end of Section 3.2.1) in which the objects o_1 and o_2 are respectively either $H_1[\ell]$ and $H_2[\ell]$, or the empty object when undefined. Note how Rules RED-FIELD-ASN-2($\mathtt{f}$) and RED-DELETE-1($\mathtt{f}$) test whether ℓ is in the current heap. The invariant that referred locations are associated to an object is not ensured by types, and rules whose transfer functions need $H[\ell]$ to be defined check whether ℓ is indeed present in the heap H. If it is not the case (the pointer being invalid), the construction of a concrete derivation gets stuck.

This language—which we call O'WHILE—is simple compared to JAVASCRIPT, with only 41 semantic rules. There are no special fields in objects—in particular no implicit prototype (see Section 1.2.3)—, nor special values with implicit type conversions (see Section 1.2.4). The memory model of JAVASCRIPT is close, though. In particular, we believe that every analyses targeting this language (and based on the techniques presented in Chapter 4) can be translated to an analysis targetting JAVASCRIPT. Indeed, the rules of the O'WHILE language have been chosen to expose difficult aspects of the rules of JSCERT. Most of the rules of JSCERT are relatively simple: the difficulty associated with JSCERT is its size, no its inherent complexity. Building an abstract version of each rule of JSCERT will be significantly long, but we do not expect any major difficulty. Alternatively, the O'WHILE language has also been designed to be very close to the pseudo-JAVASCRIPT of JSEXPLAIN (see Section 2.8). As a consequence, it is (at least in theory) possible to analyse a JAVA-

$$\textsc{red-new-obj}\quad \frac{\ell = \mathit{fresh}\left(H\right)}{H, H_e, \ell_e, \ell_c, \mathit{alloc} \Downarrow H\left[\ell \leftarrow \{\}\right], H_e, \ell_e, \ell}$$

$$\textsc{red-field}(e, \mathtt{f})\quad \frac{H, H_e, \ell_e, \ell_c, e \Downarrow r \qquad r, .\mathtt{f} \Downarrow r'}{H, H_e, \ell_e, \ell_c, e.\mathtt{f} \Downarrow r'}$$

$$\textsc{red-field-1}(\mathtt{f})\quad \frac{}{\left(H, H_e, \ell_e, \ell\right), .\mathtt{f} \Downarrow H, H_e, \ell_e, H\left[\ell\right]\left[\mathtt{f}\right]} \quad \ell.\mathtt{f} \in \mathit{dom}^2\left(H\right)$$

$$\textsc{red-in}(\mathtt{f}, e)\quad \frac{H, H_e, \ell_e, \ell_c, e \Downarrow r \qquad r, \mathtt{f}\ \mathit{in}_1 \cdot \Downarrow r'}{H, H_e, \ell_e, \ell_c, \mathtt{f}\ \mathit{in}\ e \Downarrow r'}$$

$$\textsc{red-in-1-true}(\mathtt{f})\quad \frac{}{\left(H, H_e, \ell_e, \ell\right), \mathtt{f}\ \mathit{in}_1 \cdot \Downarrow H, H_e, \ell_e, 1} \quad \ell.\mathtt{f} \in \mathit{dom}^2\left(H\right)$$

$$\textsc{red-in-1-false}(\mathtt{f})\quad \frac{}{\left(H, H_e, \ell_e, \ell\right), \mathtt{f}\ \mathit{in}_1 \cdot \Downarrow H, H_e, \ell_e, 0} \quad \ell.\mathtt{f} \notin \mathit{dom}^2\left(H\right)$$

$$\textsc{red-field-asn}(e_1, \mathtt{f}, e_2)\quad \frac{H, H_e, \ell_e, \ell_c, e_1 \Downarrow r \qquad \ell_c, r, .\mathtt{f} :=_1 e_2 \Downarrow r'}{H, H_e, \ell_e, \ell_c, e_1.\mathtt{f} := e_2 \Downarrow r'}$$

$$\textsc{red-field-asn-1}(\mathtt{f}, e_2)\quad \frac{H, H_e, \ell_e, \ell_c, e_2 \Downarrow r \qquad \ell_c, \ell, r, .\mathtt{f} :=_2 \cdot \Downarrow r'}{\ell_c, \left(H, H_e, \ell_e, \ell\right), .\mathtt{f} :=_1 e_2 \Downarrow r'}$$

$$\textsc{red-field-asn-2}(\mathtt{f})\quad \frac{o = H\left[\ell\right] \qquad H' = H\left[\ell \leftarrow o\left[\mathtt{f} \leftarrow v\right]\right]}{\ell_c, \ell, \left(H, H_e, \ell_e, v\right), .\mathtt{f} :=_2 \cdot \Downarrow H', H_e, \ell_e, \ell_c} \quad \ell \in \mathit{dom}\left(H\right)$$

$$\textsc{red-delete}(e, \mathtt{f})\quad \frac{H, H_e, \ell_e, \ell_c, e \Downarrow r \qquad \ell_c, r, \mathit{delete}_1\ .\mathtt{f} \Downarrow r'}{H, H_e, \ell_e, \ell_c, \mathit{delete}\ e.\mathtt{f} \Downarrow r'}$$

$$\textsc{red-delete-1}(\mathtt{f})\quad \frac{o = H\left[\ell\right] \qquad H' = H\left[\ell \leftarrow o \setminus \mathtt{f}\right]}{\ell_c, \left(H, H_e, \ell_e, \ell\right), \mathit{delete}_1\ .\mathtt{f} \Downarrow H', H_e, \ell_e, \ell_c} \quad \ell \in \mathit{dom}\left(H\right)$$

Figure 6.3: Rules added to manipulate the heap

Script program by analysing the JSExplain interpreter on this JavaScript program. This is similar to approaches using λ_{JS} as an intermediate language to analyse JavaScript programs [VM11b]. Because of the several layers of abstraction, results might not be as precise as a direct approach, but domains can be adapted to this end. This language is thus a good target towards a certified analysis of the full JavaScript language. We now consider how separation logic works in this setting.

6.2 About Separation Logic

Separation logic [IO01; ORY01; Rey02; Rey08] aims at precisely abstracting the heap. It was original designed to model the sharing of resources in a parallel setting, making sure that no two threads can write in the same memory cell at the same time. Separation logic lead to the construction of successful tools able to analyse complex heap structures [BCO05; Cal+09]. Fortunately for us, there is no threads in JavaScript (or at least in ECMAScript 6). We are here using separation logic as a powerful tool to abstract the heap and perform modular analyses.

To use separation logic, we need the concrete domain to be equipped with a structure of separation algebra [DHA09], defined by the four criteria below. In our case, this structure is given by $\uplus$. This operator is partial (it only applies when the domains dom^2 of its arguments are disjoint), cancellative (see Criterion 6.1 below), and forms a commutative monoid with the empty heap ϵ as neutral element (see Criteria 6.2 to 6.4).

$$\forall H_1, H_2, H.\ H_1 \uplus H = H_2 \uplus H \implies H_1 = H_2 \tag{6.1}$$

$$\forall H.\ H \uplus \epsilon = \epsilon \uplus H = H \tag{6.2}$$

$$\forall H_1, H_2, H_3.\ (H_1 \uplus H_2) \uplus H_3 = H_1 \uplus (H_2 \uplus H_3) \tag{6.3}$$

$$\forall H_1, H_2.\ H_1 \uplus H_2 = H_2 \uplus H_1 \tag{6.4}$$

Separation logic abstracts heaps by formulae. Examples of such formulae include *emp*, and $\exists \ell.\ \ell \overset{\mathtt{f}}{\mapsto} v$. The former represents an empty concrete heap, and the latter a heap with *exactly one* defined location ℓ, this location being associated to exactly one field `f` with value v. The abstract equivalent of $\uplus$ is written $\star$ (pronounced "star" or "separating conjunction"). Given two formulae ϕ_1 and ϕ_2, the formula $\phi_1 \star \phi_2$ represents a heap H which can be disjointly separated into two subheaps H_1 and H_2, each being represented by the corresponding subformula ϕ_i. In particular the formula $\ell \overset{\mathtt{f}}{\mapsto} v_1 \star \ell \overset{\mathtt{f}}{\mapsto} v_2$ does not represent any concrete heap—even when $v_1 = v_2$—as both sides of the star have non-disjoint domains. This disjointness of the star can be used to infer that some locations are different—for instance the formula $\ell_1 \overset{\mathtt{f}}{\mapsto} v_1 \star \ell_2 \overset{\mathtt{f}}{\mapsto} v_2$ implies that $\ell_1 \neq \ell_2$.

The grammar of formulae in a simple separation logic is shown below.

$$\phi ::= \phi_1 \star \phi_2 \mid \exists \ell.\ \phi \mid emp \mid \ell \overset{\mathtt{f}}{\mapsto} v$$

Formulae are related to concrete heaps by the following concretisation function γ.

$$\begin{aligned}
\gamma(\phi_1 \star \phi_2) &= \{H_1 \uplus H_2 \mid H_1 \in \gamma(\phi_1), H_2 \in \gamma(\phi_1)\} \\
\gamma(\exists \ell.\ \phi) &= \bigcup_{\ell' \in \mathcal{L}} \gamma(\phi[\ell'/\ell]) \\
\gamma(emp) &= \{\epsilon\} \\
\gamma\left(\ell \overset{\mathtt{f}}{\mapsto} v\right) &= \{\{\ell \mapsto \{\mathtt{f} : v\}\}\}
\end{aligned}$$

The concretisation of formula is invariant by commutativity and associativity of the separating conjunction $\star$, as well as the neutrality of *emp*. Separation formulae are thus considered modulo these properties by the equivalence relation $\overset{\star}{=}$. Separation logic is based on Hoare logic [Hoa69; Flo67]. As such, separation logic statements consist of semantic triples of the form $\phi, t \Downarrow \phi'$, specifying the pre-condition ϕ and post-condition ϕ' on the state of the heap during the evaluation of the term t.

To be able to analyse programs, various approximations are usually added to the grammar above. These approximations can be simple and general [BCI11] or specific to the kind of structures used by the analysed programs [GMS12]. A usual approximation is the list predicate. Consider for instance the following augmented grammar of formulae.

$$\phi ::= \dots \mid \phi_1 \vee \phi_2 \mid \mathit{list}\,(\ell)$$

We define the concretisation of these new formulae as follows. The concretisation of $\mathit{list}\,(\ell)$ is the set of all heaps only containing a linked list starting from the location ℓ.

$$\gamma\,(\phi_1 \vee \phi_2) = \gamma\,(\phi_1) \cup \gamma\,(\phi_2)$$
$$\gamma\,(\mathit{list}\,(\ell)) = \{\{\ell \mapsto \{\mathtt{next} : \ell_1\},\, \ell_1 \mapsto \{\mathtt{next} : \ell_2\},\, \dots,\, \ell_n \mapsto \{\mathtt{next} : 0\}\} \mid \ell_1, \dots, \ell_n \in \mathcal{L}\}$$

The $\vee$-construct is simple in its definition, but is difficult to manipulate. In particular, a formula such as $\mathit{emp} \vee \ell \overset{\mathtt{f}}{\mapsto} v$ can not be associated with any notion of domains, as the field `f` of location ℓ might or might not be specified by this formula. The concretisation of the *list* predicate is defined such that $\mathit{list}\,(\ell)$ is equivalent to $\ell \xmapsto{\mathtt{next}} 0 \vee \exists \ell'.\, \ell \xmapsto{\mathtt{next}} \ell' \star \mathit{list}\,(\ell')$. More general structure can be defined. For instance, Brotherston and Gorogiannis [BG14] proposed a method to handle arbitrary inductively defined constructs in separation logic. These additional constructs can make separation logic difficult to manipulate [BCO04], in particular about when approximation should occur. The fragment of the logic used in analysers [BCO05; CDV09] is usually relatively simple—typically limited to lists or trees.

What makes the modularity of separation logic is the frame rule, shown below. It states that when analysing the actions of a term on the heap, we can focus on some parts of the current formula. In other words, if enough information is present in the formula ϕ to build a derivation for a term t, then adding a context ϕ_c will not change the actions of t.

$$\textsc{frame}\quad \frac{\phi, t \Downarrow \phi'}{\phi \star \phi_c, t \Downarrow \phi' \star \phi_c}$$

We have already encountered the similar rule FRAME-ENV in Section 5.2.2. The frame rule enables abstract rules to only focus on local actions [COY07]. For instance, we can specify the abstract version of Rule RED-SKIP to only apply on the *emp* formula. From this simple abstract rule—which is simple to prove sound—, we can build a derivation from any other formula using the frame rule, as shown below.

$$\dfrac{\dfrac{}{\mathit{emp}, \mathit{skip} \Downarrow \phi'}\,\textsc{red-skip}}{\mathit{emp} \star \phi, \mathit{skip} \Downarrow \mathit{emp} \star \phi}\,\textsc{frame}$$

The frame rule is usually annotated with a side-condition about the variables modified during the execution of the term t. Formulae can indeed serve to also abstract the environment, which is usually considered to be of a different kind of object than the heap. Such formalisations differentiate *spatial* formulae, which are about the heap, and *pure* formulae, which states properties about the environment. In this work, we chose to consider variables as spatial resources [PBC06], which enables us to remove this additional condition. Section 6.3.2 shows how environments are abstracted in this setting.

This chapter presents how separation logic can be used in our framework to analyse programs of the O'WHILE language. We start by defining some abstract domains.

6

6.3 Abstract Domains

Separation logic manipulates statements of the form $\phi, t \Downarrow \phi'$. These Hoare triples seem to fit the basic restrictions of our pretty-big-step formalism. This section presents how the different semantic elements of the concrete domains can be translated into formulae.

6.3.1 Abstract Formulae

We introduced a new kind of concrete value in Section 6.1: locations $\ell \in \mathcal{L}$. To abstract them, we could use existentials as in the previous section, but they can be difficult to use. We restrict formulae such that existential quantifiers stand outmost of formulae: formulae are of the form $\exists \ell_1, \dots, \ell_n.\ \phi$, where ϕ does not contain any existential quantifier. Even in this form, some properties can be difficult to catch. Consider for instance the formula $\exists \ell_1, \ell_2.\ \ell_1 \overset{\mathtt{f}}{\mapsto} \ell_2$. There are two kinds of heap in its concretisation: heaps with a location ℓ_1 pointing through the field f to another location ℓ_2 (to which is not associated any object), and heaps with a location ℓ pointing to itself. It can be difficult to know which operations are safe on this formula. To simplify the definition and proof of transfer functions, we further restrict formulae to the form below, in which ϕ is an existential-free formula.

$$\exists \ell_1, \dots, \ell_n.\ \left(\bigstar_{i \neq j} \ell_i \neq \ell_j \right) \star \phi$$

This form makes explicit that the locations ℓ_i used in the formula are all different. For the sake of readability, we use instead an alternative but equivalent definition. We consider abstract locations $l \in L$—note the typographical differences with concrete locations $\ell \in \mathcal{L}$. Formulae are defined by the grammar below (without any constraints): abstract locations are implicitly considered existentially quantified outmost of their formula.

$$\phi ::= \phi_1 \star \phi_2 \mid \mathit{emp} \mid l \overset{\mathtt{f}}{\mapsto} v$$

We define the concretisation of these formulae in two steps. We first chose which concrete location ℓ each abstract location l represents, second the entailment predicate $\vDash$ proceeds inductively on the structure of the formula. The choice of the concrete value of abstract locations is given by a valuation $\rho : L \rightharpoonup \mathcal{L}$. This valuation is partial, only defined on the abstract locations used by the formula—if not, the entailment predicate will fail to produce a concretisation. The rules of the entailment are shown below. The concretisation function is defined as $\gamma(\phi) = \{(E, H) \mid \exists \rho.\ (E, H) \vDash_\rho \phi\}$.

$$\frac{}{(\epsilon, \epsilon) \vDash_\rho \mathit{emp}} \qquad \frac{}{(\epsilon, \{\rho(l) \mapsto \{\mathtt{f} : v\}\}) \vDash_\rho l \overset{\mathtt{f}}{\mapsto} v}$$

$$\frac{(E_1, H_1) \vDash_\rho \phi_1 \qquad (E_2, H_2) \vDash_\rho \phi_2}{(E_1 \uplus E_2, H_1 \uplus H_2) \vDash_\rho \phi_1 \star \phi_2}$$

Section 6.3.4 provides a more precise abstraction of formulae based on abstract values. We first consider how to abstract values, environments, and objects.

6.3.2 Abstract Values and Environments

Instead of tracking precise values as shown above, we can introduce abstract domains to the formalism of separation logic. We reuse the abstract values defined in Section 4.7.1, and add abstract locations to the poset of abstract values, as shown below. We update the definition of $\mathit{Store}^\sharp$ to make it more precise: each store value $u^\sharp$ is represented as a pair of a value $v^\sharp$ and a flag indicating whether it is defined $\square$ or may be undefined $\boxtimes$. The store value $\left(v^\sharp, \square\right)$ represents a defined store value whose value is abstracted by $v^\sharp$, whilst $\left(v^\sharp, \boxtimes\right)$ represents a potentially undefined store value whose value is abstracted by $v^\sharp$ if defined. This is exactly the product poset (Definition 3.2) between $\mathit{Val}^\sharp$ and $\{\square, \boxtimes\}$, where $\square \sqsubseteq \boxtimes$. Indeed, $\boxtimes$ does not mean that the property is undefined, but that the considered field *may* be undefined, and thus describes more behaviours than $\square$.

$$\begin{aligned} v^\sharp \in \mathit{Val}^\sharp &= \mathit{Sign}^\sharp \times \mathcal{P}(L) \times \mathcal{C}^\sharp \\ u^\sharp \in \mathit{Store}^\sharp &= \mathit{Val}^\sharp \times \{\square, \boxtimes\} \\ \mathit{clo} \in \mathcal{C}^\sharp &= \mathcal{P}\left(\mathit{Var} \times \mathit{Stat} \times \mathcal{L}_e^\sharp\right) \\ E^\sharp \in \mathit{Env}^\sharp &= \mathit{Var} \rightarrow \mathit{Store}^\sharp \end{aligned}$$

As indicated in Section 3.5.1, we use coercions to shorten notations. Each basic value $b^\sharp \in \mathit{Sign}$ is coerced to the abstract value $\left(b^\sharp, \varnothing, \varnothing\right) \in \mathit{Val}^\sharp$, each abstract location $l \in L$ to $(\bot, \{l\}, \varnothing) \in \mathit{Val}^\sharp$, and each closure $(\eta, \lambda \mathsf{x}.s)$ to $(\bot, \varnothing, \{(\eta, \lambda \mathsf{x}.s)\}) \in \mathit{Val}^\sharp$. Similarly, each abstract value $v^\sharp \in \mathit{Val}^\sharp$ are coerced to the store value $\left(v^\sharp, \square\right) \in \mathit{Store}^\sharp$, and $\boxtimes$ is coerced to $(\bot, \boxtimes) \in \mathit{Store}^\sharp$. As an example, the store value $u^\sharp = + \sqcup l_1 \sqcup l_2 \sqcup \boxtimes$ is equal to $((+, \{l_1, l_2\}, \varnothing), \boxtimes)\} \in \mathit{Store}^\sharp$ and represents a potentially undefined store value whose value may be a positive number, or the location represented by either l_1 or l_2.

Abstract closures carry environment locations $\eta \in \mathcal{L}_e^\sharp$ to represent environment locations, as in Section 4.7.1. Closures could have been abstracted to carry environments, but Coq imposes some restrictions on the way terms should be defined to prevent terms from looping. This restriction makes such a direct approach difficult: we would have to enforce the number of defined variable to decrease along the environment structure, which is possible but difficult to enforce [AR99]. We use an indirection through environments locations: each abstract environment is identified by an environment identifier $\eta \in \mathcal{L}_e^\sharp$. Environments are specified in formulae using predicates such as $\eta \mapsto \{\mathsf{x} \mapsto v^\sharp\}$. The closure $(\eta, \lambda\mathsf{x}.s)$ points to the environment referred by η, in which the statement s will be executed. The global environment and the context environment (see Section 4.7.1) are represented by two abstract environment locations η carried by the formula.

As in Definition 3.2 of the product poset, the concretisation of values and store values are the union of the concretisations of their components. Because of abstract locations, The concretisation of a value depends on a valuation $\rho_L : L \rightharpoonup \mathcal{P}(\mathcal{L})$. The codomain of this valuation is the powerset of concrete locations, but we enforce $\rho_L(l)$ to be a singleton for all location l. This formalisation choice simplifies notations, in particular in Section 6.5 in which this constraint is weakened. Similarly, because of closures, we need a valuation $\rho_E : \mathcal{L}_e^\sharp \to \mathcal{P}(\mathcal{L}_e)$. We thus consider a valuation $\rho : (L \rightharpoonup \mathcal{P}(\mathcal{L})) \uplus (\mathcal{L}_e^\sharp \to \mathcal{P}(\mathcal{L}_e))$. The concretisation γ_ρ of the abstract value $v^\sharp = (b^\sharp, \Lambda, clo)$ is defined below. We recall that $\dot{\rho}(\Lambda) = \bigcup_{l \in \Lambda} \rho(l)$, following the notation defined in Section 6.1.

$$\gamma_\rho\left(\left(b^\sharp, \Lambda, clo\right)\right) = \gamma\left(b^\sharp\right) \uplus \dot{\rho}(\Lambda) \uplus \{(\ell_c, \lambda\mathsf{x}.s) \mid \ell_c \in \rho(\eta), (\eta, \lambda\mathsf{x}.s) \in clo\}$$

The concretisations of store values are subset of $Val \uplus \{undef\}$. The special value $undef$ denotes that the considered store value may be undefined. This concretisation is defined as the concretisation of the product poset (see Section 3.5.1), where $\gamma_\rho(\boxtimes) = \{undef\}$ and $\gamma_\rho(\square) = \varnothing$. The concretisation of abstract environments follows. Note how concrete environments are finite maps to Val (see Definition 3.1), but abstract environments are complete maps to $Store^\sharp$, which represents both Val and the special value $undef$.

$$E \in \gamma_\rho\left(E^\sharp\right) \iff \forall \mathsf{x}. \begin{cases} \mathsf{x} \in dom(E) \implies E[\mathsf{x}] \in \gamma_\rho\left(E^\sharp[\mathsf{x}]\right) \\ \mathsf{x} \notin dom(E) \implies undef \in \gamma_\rho\left(E^\sharp[\mathsf{x}]\right) \end{cases}$$

As said in the previous section, our formalisation relies on the hypothesis that in a given formula two locations with different names represent different concrete locations. The traditional separation logic formula $\exists \ell_2.\ \ell_1 \mapsto \{\mathsf{f} : \ell_2\}$ expressed that the field f of the location ℓ_1 points to either ℓ_1 or to another location. In our formalism, we would write such a formula in the form $l_1 \mapsto \{\mathsf{f} : l_1 \sqcup l_2\}$ where l_2 is another location to which we suspect the field f to point—the abstract value $l_1 \sqcup l_2$ being equal to $(\bot, \{l_1, l_2\}, \varnothing)$. The uncertainty has been shifted from the location level to the value level: it is now clear that the two locations l_1 and l_2 are distinct, but we allow values to be imprecise.

6.3.3 Abstract Objects

As for JavaScript objects, the objects of the O'While language of Section 6.1 are extensible. In particular, there are cases in which it is important to precisely know whether a given field $\mathtt{f}$ is present in an object—typically in the presence of the $\mathtt{f}\ in\ e$ operator. It is thus natural to use abstract store values $u^\sharp \in Store^\sharp$ to account for the values of object fields, using the construct $l \overset{\mathtt{f}}{\mapsto} u^\sharp$. However, this construct only accounts for one field. To precisely abstract a newly allocated object—whose fields are all undefined—the size of a formula would have to be infinite. We are thus in the need of an abstraction of objects.

Conceptually, abstract objects $o^\sharp$ are partial maps from fields to abstract store values $Field \rightharpoonup Store^\sharp$. The domain of an object is the specified domain of this object. For instance, a partial map $o^\sharp$ undefined on all fields except $\mathtt{f}$, which is mapped to $u^\sharp$, does not specify the field $\mathtt{g}$—in particular, the frame rule can add a specified field $\mathtt{g}$ in such an object. On the contrary, the map mapping $\mathtt{f}$ to $u^\sharp$ and every other fields to $\boxtimes$ specifies that the field $\mathtt{g}$ of this object is undefined: the frame rule can not add this field.

Partial maps from fields to store values $Field \rightharpoonup Store^\sharp$ are too precise, but follow the intuition behind the abstraction of objects. We instead consider two kinds of abstract objects: *finite* and *cofinite* objects. Finite objects are objects specified on a finite domain. As for concrete objects, we use a JavaScript-like notation: an abstract object mapping $\mathtt{f}$ to $u_1^\sharp$ and $\mathtt{g}$ to $u_2^\sharp$ is written $\left\{\mathtt{f} : u_1^\sharp,\ \mathtt{g} : u_2^\sharp\right\}$. Cofinite objects represent partial functions defined for *all but a finite set* F of fields, that is, their domain is a cofinite set in $Field$. In the simplest case, a cofinite object assigns one abstract value $u^\sharp$ for all fields F specified by the object. We write such an object $\left\{\overline{F} : u^\sharp\right\}$. We write explicitly the set of fields $F \subseteq Field$ on which the object is *not* specified. More generally, cofinite objects have the form below, where $\{\mathtt{f}_1, \dots, \mathtt{f}_n\} \subseteq F$. This last constraint helps preventing conflicts between field values—it could be removed by stating that the left-most field declaration have priority.

$$\left\{\mathtt{f}_1 : u_1^\sharp,\ \dots,\ \mathtt{f}_n : u_n^\sharp,\ \overline{F} : u^\sharp\right\}$$

In this form, we list specific abstract values for a finite set of fields. For example, the abstract object $\left\{\mathtt{f} : \boxtimes,\ \mathtt{g} : u^\sharp,\ \overline{\{\mathtt{f}, \mathtt{g}\}} : \top\right\}$ describes the set of objects whose field $\mathtt{f}$ is absent, field $\mathtt{g}$ can be abstracted by $u^\sharp$, and any other fields can be anything (including undefined). As a convenient shorthand, fully specified objects are written $\left\{\mathtt{f}_1 : u_1^\sharp,\ \dots,\ \mathtt{f}_n : u_n^\sharp,\ \_ : u^\sharp\right\}$, the symbol _ standing for $\overline{\{\mathtt{f}_1, \dots, \mathtt{f}_n\}}$. The frame rule can not complete a fully specified object. The abstract object $\{\_ : \boxtimes\}$ thus describes the fully specified empty object, for instance returned by the instruction $alloc$.

We write $spec\left(o^\sharp\right)$ for the set of fields which the abstract object $o^\sharp$ specifies.

$$spec\left(\left\{\mathtt{f}_1 : u_1^\sharp,\ \dots,\ \mathtt{f}_n : u_n^\sharp\right\}\right) = \{\mathtt{f}_1, \dots, \mathtt{f}_n\}$$
$$spec\left(\left\{\mathtt{f}_1 : u_1^\sharp,\ \dots,\ \mathtt{f}_n : u_n^\sharp,\ \overline{F} : u^\sharp\right\}\right) = \{\mathtt{f}_1, \dots, \mathtt{f}_n\} \uplus (Field \setminus F)$$

We write $o^\sharp[\texttt{f}]$ for $\texttt{f} \in \mathit{spec}\left(o^\sharp\right)$ the abstract value associated with $\texttt{f}$ in the abstract object $o^\sharp$, and $o^\sharp\left[\texttt{f} \leftarrow u^\sharp\right]$ an object similar to $o^\sharp$ expect that $\texttt{f}$ is mapped to $u^\sharp \in \mathit{Store}^\sharp$.

We extend the partial order to abstract objects as the point-wise order on the underlying partial functions, with the added constraint that the two objects must specify the same fields. There is thus no abstract object greater than any other abstract object. If we restrict ourself to a given specification domain F, though, the restriction of the object poset forms a complete lattice whose greatest element is the object mapping every field $\texttt{f} \in F$ to $\top$.

$$o_1^\sharp \sqsubseteq o_2^\sharp \iff \begin{cases} \mathit{spec}\left(o_1^\sharp\right) = \mathit{spec}\left(o_2^\sharp\right) \\ \forall \texttt{f} \in \mathit{spec}\left(o_1^\sharp\right).\ o_1^\sharp[\texttt{f}] \sqsubseteq o_2^\sharp[\texttt{f}] \end{cases}$$

The concretisation of objects is defined similarly to the concretisation of environments, except that abstract objects are partial whilst abstract environments are total. In particular, this concretisation is meant to be expandable, as expressed by Proposition 6.1 below.

$$o \in \gamma_\rho\left(o^\sharp\right) \iff \forall \texttt{f}. \begin{cases} \texttt{f} \in \mathit{dom}(o) \implies \texttt{f} \in \mathit{spec}\left(o^\sharp\right) \wedge o[\texttt{f}] \in \gamma_\rho\left(o^\sharp[\texttt{f}]\right) \\ \texttt{f} \notin \mathit{dom}(o) \implies \texttt{f} \in \mathit{spec}\left(o^\sharp\right) \vee \mathit{undef} \in \gamma_\rho\left(o^\sharp[\texttt{f}]\right) \end{cases}$$

Proposition 6.1 states that the concretisation of abstract objects is compatible with their domain. It will be used to cut objects in formulae, by stating that a formula of the form $l \mapsto o_1^\sharp \uplus o_2^\sharp$ is equivalent to $l \mapsto o_1^\sharp \star l \mapsto o_2^\sharp$, the two objects specifying different fields.

Proposition 6.1. *The concretisation of abstract objects is such that for all* $o_1 \in \gamma_\rho\left(o_1^\sharp\right)$ *and* $o_2 \in \gamma_\rho\left(o_2^\sharp\right)$ *such that* $\mathit{spec}\left(o_1^\sharp\right) \cap \mathit{spec}\left(o_2^\sharp\right) = \varnothing$*, we have* $o_1 \uplus o_2 \in \gamma_\rho\left(o_1^\sharp \uplus o_2^\sharp\right)$*.*

6.3.4 Abstract State Formulae

Now that the abstractions of values, environments, and objects have been defined, we can formally define the grammar of formulae as below.

$$\begin{aligned} \phi &::= \mathit{emp} \mid \phi_1 \star \phi_2 \mid \eta \mapsto E^\sharp \mid l \mapsto o^\sharp \\ o^\sharp &::= \left\{\texttt{f}_1 : u_1^\sharp, \ldots, \texttt{f}_n : u_n^\sharp\right\} \mid \left\{\texttt{f}_1 : u_1^\sharp, \ldots, \texttt{f}_n : u_n^\sharp, \overline{F} : u^\sharp\right\} \end{aligned}$$

To further simplify definitions, we write p for pointers (l or η), and $œ^\sharp$ for pointees ($o^\sharp$ or $E^\sharp$). We write $p \mapsto œ^\sharp \in \phi$ if $p \mapsto œ^\sharp$ occurs in ϕ. The set $P(\phi)$ is defined as $\left\{p \mid p \mapsto œ^\sharp \in \phi\right\}$, and the set $Œ(\phi)$ is defined as $\left\{œ^\sharp \mid p \mapsto œ^\sharp \in \phi\right\}$.

We define an equivalence relation $\stackrel{\star}{=}$ on formulae as the smallest relation such that $\star$ is associative, commutative, has emp as neutral element, and respects object splitting—that is, if $\mathit{spec}\left(o_1^\sharp\right) \cap \mathit{spec}\left(o_2^\sharp\right) = \varnothing$, then $l \mapsto o_1^\sharp \star l \mapsto o_2^\sharp \stackrel{\star}{=} l \mapsto o_1^\sharp \uplus o_2^\sharp$. Environments are always fully specified and can not be split: for any environment $E^\sharp$, we define $\mathit{spec}\left(E^\sharp\right) = \mathit{Var}$.

$$\frac{}{(\epsilon, \varnothing, \epsilon) \vDash_\rho \mathbf{emp}} \qquad \frac{(H_{e,1}, D_1, H_1) \vDash_\rho \phi_1 \qquad (H_{e,2}, D_2, H_2) \vDash_\rho \phi_2}{(H_{e,1} \uplus H_{e,2}, D_1 \uplus D_2, H_1 \uplus H_2) \vDash_\rho \phi_1 \star \phi_2}$$

$$\frac{\rho(l) = \{\ell\} \qquad o \in \gamma_\rho\left(o^\sharp\right)}{\left(\epsilon, \{\ell\} \times spec\left(o^\sharp\right), \ell \mapsto o\right) \vDash_\rho l \mapsto o^\sharp} \qquad \frac{\ell_e \in \rho(\eta) \qquad E \in \gamma_\rho\left(E^\sharp\right)}{(\{\ell_e \mapsto E\}, \varnothing, \epsilon) \vDash_\rho \eta \mapsto E^\sharp}$$

Figure 6.4: Definition of the entailment predicate $\vDash_\rho$

The concretisation of formulae depends on the entailment defined in Figure 6.4. The entailment $\vDash_\rho$ states when a concrete state (H_e, D, H) is abstracted by a formula ϕ. To account for the partiality of objects (see previous section), the entailment uses a specification domain $D \subseteq \mathcal{L} \times Field$. For instance, the specification domain of $l \mapsto \{\mathtt{f} : \boxtimes\}$ is $\{\ell, \mathtt{f}\}$ when $\rho(l) = \{\ell\}$: any pair of location and field in $D \setminus dom^2(H)$ is specified but undefined. This enforces the formula $l \mapsto \{\mathtt{f} : \boxtimes\} \star l \mapsto \{\mathtt{f} : \boxtimes\}$ to have an empty concretisation, as the two definition domains of the concretisations of the two subformulae intersect. The specification domain follows the invariant $dom^2(H) \subseteq D$. As for values, environments, and objects, the entailment is parametrised by a valuation $\rho : (L \rightharpoonup \mathcal{P}(\mathcal{L})) \uplus \left(\mathcal{L}_e^\sharp \rightharpoonup \mathcal{P}(\mathcal{L}_e)\right)$. The concretisation of formulae is then defined as below.

$$\gamma(\phi) = \{(H_e, H) \mid \exists \rho, D.\ (H_e, D, H) \vDash_\rho \phi\}$$

The entailment has been defined to be compatible with the $\stackrel{\star}{=}$ relation.

Proposition 6.2. *For all two formulae ϕ_1 and ϕ_2 such that $\phi_1 \stackrel{\star}{=} \phi_2$, we have the equivalence $(H_e, D, H) \vDash_\rho \phi_1 \iff (H_e, D, H) \vDash_\rho \phi_2$. In particular, we have $\gamma(\phi_1) = \gamma(\phi_2)$.*

The frame rule enables formulae to be extended. But there are some properties which we want to check before extending formulae. In particular, we have seen that the formula $l \mapsto \{\mathtt{f} : \boxtimes\} \star l \mapsto \{\mathtt{f} : \boxtimes\}$ has an empty concretisation because of conflicting domains. Formula domains are subsets of $L \times Field \uplus \mathcal{L}_e^\sharp \times Var$. The *interface* $itf(\phi) \in L \uplus \mathcal{L}_e^\sharp$ of a formula ϕ is the set of abstract locations l and environment locations η appearing in ϕ.

Definition 6.1. The domain and interface of formulae are defined as follows.

$$\begin{array}{ccc} dom(\phi_1 \star \phi_2) = dom(\phi_1) \cup dom(\phi_2) & dom(\mathbf{emp}) = \varnothing & dom\left(p \mapsto œ^\sharp\right) = \{p\} \times spec\left(œ^\sharp\right) \\ itf(\phi_1 \star \phi_2) = itf(\phi_1) \cup itf(\phi_2) & itf(\mathbf{emp}) = \varnothing & itf\left(p \mapsto œ^\sharp\right) = \{p\} \cup itf\left(œ^\sharp\right) \end{array}$$

The interface $itf \in L \uplus \mathcal{L}_e^\sharp$ of an object $o^\sharp$, an abstract environment $E^\sharp$, an abstract value $v^\sharp = \left(b^\sharp, \Lambda, clo\right)$, and an abstract store value $u^\sharp = \left(v^\sharp, d\right)$ (with $d \in \{\square, \boxtimes\}$) is the set of abstract locations l and environment identifiers η appearing in them.

Definition 6.2. The interface of abstract objects, abstract environments, abstract values, and abstract store values are defined as follows.

$$itf\left(o^\sharp\right) = \bigcup_{\mathsf{f} \in spec(o^\sharp)} itf\left(o^\sharp[\mathsf{f}]\right) \qquad itf\left(E^\sharp\right) = \bigcup_{\mathsf{x} \in Var} itf\left(E^\sharp[\mathsf{x}]\right)$$

$$itf\left(b^\sharp, \Lambda, clo\right) = \Lambda \uplus \{\eta \mid \exists \mathsf{x}, s.\ (\eta, \lambda \mathsf{x}.s) \in clo\} \qquad itf\left(\left(v^\sharp, d\right)\right) = itf\left(v^\sharp\right)$$

We have defined formulae precisely modelling the heap and the environments. But states in an abstract derivation may carry other kinds of values. The next section presents how formulae can be extended to take these values into account.

6.3.5 Extended Formulae

The pretty-big-step format introduces intermediary semantic contexts σ (see Section 2.1.1). For instance, the intermediary term `stat_while_1 L e t rv` of Section 2.5.2.1 carries a value `rv`. This is especially visible in the dependent version of pretty-big-step of Section 4.5, in which the intermediary term $\cdot +_1 e_2$ expects a semantic context in $Env \times Out_e$ (see Figure 4.11a). These extended semantic contexts are abstracted by extended formulae, which are pairs of a formula ϕ and a carried extension $x^\sharp$, whose type depends on the associated intermediary term. Importantly, the locations of the extension are linked with the locations of the formula. Consider for instance that the carried extension $x^\sharp$ is an abstract location, then the extended formula $(l_1 \mapsto \{\mathsf{f} : l_1\}, l_1)$ has a different concretisation than $(l_1 \mapsto \{\mathsf{f} : l_1\}, l_2)$. We translates this by making the entailment of the formula share the valuation ρ with the concretisation of the extension, as shown below.

$$\gamma\left(\left(\phi, x^\sharp\right)\right) = \left\{(H_e, H, x) \;\middle|\; \exists \rho, D.\ (H_e, D, H) \vDash_\rho \phi \wedge x \in \gamma_\rho\left(x^\sharp\right)\right\}$$

In the context of our O'While language, semantic contexts for expressions and statements already carry additional information (see Figure 6.3). The concrete semantic contexts of these terms are in the form H, H_e, ℓ_e, ℓ_c. The two heaps H and H_e translate into a formula. The two environment locations ℓ_e and ℓ_c translates into environment locations η. A basic example of abstract semantic context is thus $(\eta \mapsto \epsilon, \eta, \eta)$, the extension being a pair of abstract environment locations. Each extension $x^\sharp$ has an interface, written $itf\left(x^\sharp\right)$.

The frame rule can be extended to apply on extended formulae. To this end, we extend the $\star$ operator to account for extended terms, as shown below. We suppose that at least one of the context formula ϕ_c or the main formula ϕ is a simple formula, without extension.

$$\left(\phi, x^\sharp\right) \star \phi_c = \left(\phi \star \phi_c, x^\sharp\right) \qquad \phi \star \left(\phi_c, x^\sharp\right) = \left(\phi \star \phi_c, x^\sharp\right)$$

6.4 The Frame Rule and Membranes

As we have seen in Section 6.2, the frame rule is a defining feature of separation logic. This rule enables modular reasoning. For instance, we can specify what is the behaviour of a function (given by a specific closure) by small semantic triples featuring only what the function manipulates, then reuse these triples in larger environments using the frame rule. The frame rule can also be used to remove an unwanted context from the current state formula, thus simplifying the analysis. The frame rule does not correspond to any concrete rule. As such, it is a structural rule (see Chapter 5). However, this rule comes with some issues, which we first address.

The frame rule requires extra care because of the potential collision between location names in the local state ϕ and the context ϕ_c. This may happen when abstract locations are renamed or allocated. Consider for instance the two formulae $\phi_1 = \eta \mapsto \epsilon \star l_1 \mapsto \{\mathtt{f} : l_1\}$ and $\phi_2 = \eta \mapsto \epsilon \star l_2 \mapsto \{\mathtt{f} : l_2\}$. We have $\gamma(\phi_1) = \gamma(\phi_2)$: if a valuation ρ_1 makes a concrete state (H, H_e) entail ϕ_1, we can easily build another valuation ρ_2 making it entail ϕ_2, by exchanging the concrete locations associated with l_1 and l_2. It is thus sound to weaken ϕ_1 to ϕ_2 using Rule GLUE-WEAKEN. However, when this weakening is combined with the frame rule, this may lead to an unsound result. Figure 6.5 illustrates such an interference. Its final result has an empty concretisation although we can build concrete derivations from several elements of the concretisation of the initial semantic context.

Similarly, newly allocated abstract locations must be fresh, but the frame rule can interfere with their freshness. Figure 6.6 shows a derivation in which the frame rule removes a location l from the context, leaving it empty. The abstract rule RED-NEW-OBJ then picks a fresh location from this empty abstract state. The abstract location l is fresh in the current formula, but the frame rule frames l again in the conclusion. The result also ends up having an empty concretisation, although concrete derivations can be built from several elements of the concretisation of the initial semantic context. Theorem 5.1 does not apply with the frame rule. We thus need to protect location names from the frame rule.

We introduce the notion of *membranes* as an explicit but light-weight formalism for managing these names in abstract derivations. Membranes are relations on names. For instance the membrane $(l_o \to l_i)$ relates an outer ("global") name l_o to the inner ("local") name l_i of a formula. Membraned formulae are pairs of a membrane and a formula, for instance $(l_0 \to l_1 \mid l_1 \mapsto \{\mathtt{f} : l_1\})$ describes the object pointed by the outer location l_0.

6.4.1 Membranes

We equip formulae with membranes specifying how locations from the potential context of the frame rule ("outside the membrane") are mapped to the locations of the formula ("inside the membrane"). Membranes are needed in our formalism to make the frame rule soundly interact with Rule GLUE-WEAKEN. This section formally defines membranes.

$$\dfrac{\dfrac{\overline{l_1 \mapsto \{\texttt{f} : l_1\}, \eta, \eta, \mathit{skip} \Downarrow l_1 \mapsto \{\texttt{f} : l_1\}, \eta, \eta}\ \textsc{red-skip}}{l_1 \mapsto \{\texttt{f} : l_1\}, \eta, \eta, \mathit{skip} \Downarrow l_2 \mapsto \{\texttt{f} : l_2\}, \eta, \eta}\ \textsc{glue-weaken}}{\eta \mapsto \epsilon \star l_2 \mapsto \{\texttt{f} : l_1\} \star l_1 \mapsto \{\texttt{f} : l_1\}, \eta, \eta, \mathit{skip} \Downarrow \eta \mapsto \epsilon \star l_2 \mapsto \{\texttt{f} : l_1\} \star l_2 \mapsto \{\texttt{f} : l_2\}, \eta, \eta}\ \textsc{frame}$$

Figure 6.5: Unsound interaction between Rules GLUE-WEAKEN and FRAME

$$\dfrac{\overline{\mathit{emp}, \eta, \eta, \mathit{alloc} \Downarrow l \mapsto \{\_ : \boxtimes\}, \eta, l}\ \textsc{red-new-obj}}{\eta \mapsto \epsilon \star l \mapsto \{\_ : \boxtimes\}, \eta, \eta, \mathit{alloc} \Downarrow \eta \mapsto \epsilon \star l \mapsto \{\_ : \boxtimes\} \star l \mapsto \{\_ : \boxtimes\}, \eta, l}\ \textsc{frame}$$

Figure 6.6: Unsound interaction between Rules FRAME and RED-NEW-OBJ

As hinted by the two Figures 6.5 and 6.6, there are two separate issues with the frame rule. First, the renamings performed in a formula should be kept along the computation to avoid names to be caught by the context of the frame rule. Second, newly created names have to be marked to avoid the context to inadvertently refer to them. The two derivations of Figures 6.5 and 6.6 shows the problem for abstract locations $l \in L$, but abstract environment locations $\eta \in \mathcal{L}_e^\sharp$ suffer from the same problem.

Definition 6.3. Membranes M are finite relations, defined as follows.

$$M \in \mathcal{P}_{\mathit{fin}} \left(\left(L \uplus \{\bullet\} \right) \times L \right) \uplus \mathcal{P}_{\mathit{fin}} \left(\left(\mathcal{L}_e^\sharp \uplus \{\bullet\} \right) \times \mathcal{L}_e^\sharp \right)$$

The special element $\bullet$ is an allocator: if a membrane M states that $M(\bullet, l)$, then l is a location allocated during the execution (through Rule RED-NEW-OBJ). Similarly, if a membrane M states that $M(\bullet, \eta)$, then η is a new environment location created during the execution (through a function call, in Rule RED-APP-2(s)). In the following we write $p^\bullet$ for an element of $L \uplus \mathcal{L}_e^\sharp \uplus \{\bullet\}$, $l^\bullet$ for an element of $L \uplus \{\bullet\}$, and $\eta^\bullet$ for an element of $\mathcal{L}_e^\sharp \uplus \{\bullet\}$.

We define the inner and outer interfaces In and Out of a membrane M as follows.

$$\mathit{In}(M) = \mathit{codom}(M) = \{p_i \mid \exists p_o^\bullet.\ M(p_o^\bullet, p_i)\}$$
$$\mathit{Out}(M) = \mathit{dom}(M) \smallsetminus \{\bullet\} = \{p_o \mid \exists p_i.\ M(p_o, p_i)\}$$

For readability, we write membranes as lists of atomic relations of the form $p_o^\bullet \to p_i$. We define $M(p_o^\bullet) = \{p_i \mid M(p_o^\bullet, p_i)\}$ and $M^{-1}(p_i) = \{p_o^\bullet \mid M(p_o^\bullet, p_i)\}$. In this section, we consider that membranes are functional and injective, except on $\bullet$. That is, for each p_o and p_i, $M(p_o)$ and $M^{-1}(p_i)$ are singletons. No restriction is imposed on $M(\bullet)$.

Membraned formulae Φ are pairs of a membrane and a formula. We add the constraint that the inner formula should only use local locations defined in the membrane. Membraned formulae can also carry an additional information $x^\sharp$ (see Section 6.3.5).

$$\Phi ::= \left(M \mid \phi, x^\sharp\right) \qquad\qquad itf\,(\phi) \cup itf\left(x^\sharp\right) \subseteq In\,(M)$$

We define the interface and the domain of a membraned formula as follows. The symbol `y` denotes both fields `f` and variables `x`.

$$\begin{aligned} itf\left(\left(M \mid \phi, x^\sharp\right)\right) &= Out\,(M) \\ dom\left(\left(M \mid \phi, x^\sharp\right)\right) &= \left\{(p, \mathsf{y}) \mid \exists p'.\, M\left(p, p'\right) \wedge \left(p', \mathsf{y}\right) \in dom^2\,(\phi)\right\} \end{aligned}$$

We extend the equivalence relation $\stackrel{\star}{=}$ on membraned formulae. Two membraned formulae $\Phi_1 = (M_1 \mid \phi_1)$ and $\Phi_2 = (M_2 \mid \phi_2)$ are equivalent if their inner formulae are equivalent $\phi_1 \stackrel{\star}{=} \phi_2$ and they have the same membranes $M_1 = M_2$. Membranes are compared as relations and do not depend on the order from which the rewritings $p_o^\bullet \to p_i$ are written.

The concretisation of membraned formulae $\Phi = (M \mid \phi)$ is the set of all heaps entailing the formula for a valuation. The considered valuations ρ_i have to be defined on the inner interface of the membrane M. These valuations $\rho_i : In\,(M) \rightharpoonup_{inj} \mathcal{P}\,(\mathcal{L}) \uplus \mathcal{P}\,(\mathcal{L}_e)$ must be injective: $\forall p_1, p_2.\ \rho_i\,(p_1) \cap \rho_i\,(p_2) \neq \varnothing \implies p_1 = p_2$. In other words, different abstract valuations have to represent different concrete locations. Of course, these valuations should map abstract object locations $l \in L$ to concrete $\ell \in \mathcal{L}$ and abstract environment locations $\eta \in \mathcal{L}_e^\sharp$ to concrete $\ell_e \in \mathcal{L}_e$. Furthermore, as membranes relate the outer and the inner scope, we have to check whether these relations are compatible. For instance, the membrane $(l_1 \to l_2, l_1 \to l_3)$ is not satisfiable as both l_2 and l_3 are supposed defined and different, whilst both coming from one location l_1. We thus require the existence of an outer valuation $\rho_o : Out\,(M) \rightharpoonup_{inj} \mathcal{P}\,(\mathcal{L}) \uplus \mathcal{P}\,(\mathcal{L}_e)$ related to the inner valuation ρ_i. A set of allocated locations ν is also chosen. We write $\rho_o^\nu : Out\,(M) \uplus \{\bullet\} \to_{inj} \mathcal{P}\,(\mathcal{L})$ for the function equal to ρ_o in the domain $Out\,(M)$ and such that $\rho_o^\nu\,(\bullet) = \nu$. The relation between these valuations and this set, written $(\rho_o, \nu, \rho_i) \in \gamma\,(M)$, is defined as follows.

$$\dot{\rho_o}\,(Out\,(M)) \cap \nu = \varnothing \tag{6.5}$$

$$\forall p_o^\bullet \in Out\,(M) \uplus \{\bullet\}\,.\ \rho_o^\nu\,(p_o^\bullet) \subseteq \dot{\rho_i}\,(M\,(p_o^\bullet)) \tag{6.6}$$

$$\forall p_i \in In\,(M)\,.\ \rho_i\,(p_i) \subseteq \dot{\rho_o^\nu}\left(M^{-1}\,(p_i)\right) \tag{6.7}$$

Intuitively, no concrete location is forgotten by going through the membrane, and the only new locations are the ones allocated by the membrane. For instance, there is no pair of valuations satisfying the membrane $(l_1 \to l_2, l_1 \to l_3)$ since we must have $\rho_i\,(l_2) = \rho_o\,(l_1) = \rho_i\,(l_3)$, as both valuations map abstract locations to singleton sets: the valuation ρ_i can not be injective. We finally define the concretisation of membraned formulae as follows. The extension $x^\sharp$ does not add any complexity, except that it has to use the same valuation ρ_i than the state for its concretisation.

$$(H_e, H, x) \in \gamma\left(\left(M \;\middle|\; \phi, x^\sharp\right)\right) \iff \exists \rho_i, \rho_o, \nu.\ (\rho_o, \nu, \rho_i) \in \gamma(M) \wedge (H_e, H) \vDash_{\rho_i} \phi \wedge x \in \gamma_{\rho_i}\left(x^\sharp\right) \quad (6.8)$$

6.4.2 Framing Operators

We define two basic operations on membranes: *membrane composition* $;$ and *membrane crossing* $\triangleright$. The composition of two membranes M and M' is only defined when the domains of the membranes match, that is when $Out(M') \subseteq In(M)$: the outer names of the inner membrane should be inner names of the outer membrane. It is then defined as below. It intuitively corresponds to a composition of all the rewritings $p_o^\bullet \to p_i$ present in both membranes. The allocated locations of the composition is the union of the allocated locations in both membranes, the outer locations being renamed. For instance, we have $(l_1 \to l_2, \bullet \to l_3) \,;\, (l_3 \to l_4, \bullet \to l_5) = (l_1 \to l_2, \bullet \to l_4, \bullet \to l_5)$.

$$M \,;\, M' = \left\{p^\bullet \to p' \;\middle|\; \exists p''.\ M\left(p^\bullet, p''\right) \wedge M'\left(p'', p'\right)\right\} \cup \left\{\bullet \to p' \;\middle|\; M'\left(\bullet, p'\right)\right\}$$

The membrane crossing $\triangleright$ intuitively updates all the locations in a formula using the membrane as a substitution. This operator is partial: $\phi \triangleright M$ is only defined when the interface of ϕ matches the outer domain of the membrane, that is, when $itf(\phi) \subseteq Out(M)$. Each value of the external formula ϕ also has to pass the membrane, and we define a similar operation $\triangleright$ for values. Figure 6.7 shows the rules for membrane crossing. We advise the reader to temporary ignore the complex fourth rule, which is explained in details in Section 6.5.1. Intuitively, we have $\left(p \mapsto œ^\sharp \triangleright M\right) = M(p) \mapsto \left(œ^\sharp \triangleright M\right)$. The membrane crossing of abstract objects $o^\sharp \triangleright M$ and environments $E^\sharp \triangleright M$ applies the membrane crossing on each of their values: for all y, we have $\left(œ^\sharp \triangleright M\right)[\mathsf{y}] = œ^\sharp[\mathsf{y}] \triangleright M$. Membrane crossing behaves as expected with respect to membrane composition, as shown below.

Lemma 6.3. *For any membrane M and M', and any formula ϕ, if either $(\phi \triangleright (M \,;\, M'))$ or $((\phi \triangleright M) \triangleright M')$ is defined, then the other is also defined and they are equal by $\stackrel{\star}{=}$.*

We define two partial operators $\boxed{\star}$ and $\boxed{\circ}$ on membraned formulae. They are both pronounced similarly to their corresponding symbol $\star$ and $\circ$. Figure 6.8 shows their definitions. The operator $\boxed{\star}$ imports a context (unmembraned) formula ϕ into a membraned formula Φ. It is based on the $\triangleright$ operator, and thus requires the interface of the formula ϕ to match the interface of the membraned formula Φ. The operator $\boxed{\circ}$ composes a membraned formula Φ with an external membrane M. It changes the interface of the resulting membrane: the interface of $M \boxed{\circ} \Phi$ (when it is defined) is $Out(M)$. These operators are used to define our equivalent of the frame rule. The frame rule is split into two structural

$$\frac{}{\mathit{emp} \triangleright M = \mathit{emp}} \qquad \frac{\phi \stackrel{\star}{=} \phi' \qquad \phi' \triangleright M = \phi_r}{\phi \triangleright M = \phi_r} \qquad \frac{\phi_1 \triangleright M = \phi_1' \qquad \phi_2 \triangleright M = \phi_2'}{\phi_1 \star \phi_2 \triangleright M = \phi_1' \star \phi_2'}$$

$$\frac{\dot{M}^{-1}\left(\dot{M}\left(P\left(\phi\right)\right)\right) = P\left(\phi\right) \qquad \forall\, \text{\oe}_i^\sharp, \text{\oe}'^\sharp_i \in \text{\OE}\left(\phi\right).\ \mathit{spec}\left(\text{\oe}_i^\sharp\right) = \mathit{spec}\left(\text{\oe}'^\sharp_i\right)}{\phi \triangleright M = \mathop{\bigstar}\limits_{p_j \in \dot{M}(P(\phi))} p_j \mapsto \bigsqcup_{\substack{p_i \in M^{-1}(p_j) \\ p_i \mapsto \text{\oe}_i^\sharp \in \phi}} \left(\text{\oe}_i^\sharp \triangleright M\right)}$$

$$\frac{\Lambda \in \mathcal{P}\left(L\right)}{\Lambda \triangleright M = \dot{M}\left(\Lambda\right)} \qquad \frac{}{\mathit{clo} \triangleright M = \left\{\left(\eta, \lambda \mathsf{x}.s\right) \,\middle|\, \left(\eta', \lambda \mathsf{x}.s\right) \in \mathit{clo}, \eta \in M\left(\eta'\right)\right\}}$$

$$\frac{}{\left(b^\sharp, \Lambda, \mathit{clo}\right)^\sharp \triangleright M = \left(b^\sharp, \Lambda \triangleright M, \mathit{clo} \triangleright M\right)^\sharp} \qquad \frac{}{\left(v^\sharp, d\right) \triangleright M = \left(v^\sharp \triangleright M, d\right)}$$

Figure 6.7: Rules for crossing membranes

$$\frac{\mathit{itf}\left(\phi'\right) \subseteq \mathit{Out}\left(M\right)}{\phi' \boxed{\star} \left(M \mid \phi\right) = \left(M \,\middle|\, \left(\phi' \triangleright M\right) \star \phi\right)} \qquad \frac{\mathit{Out}\left(M\right) \subseteq \mathit{In}\left(M'\right)}{M' \boxed{\odot} \left(M \mid \phi\right) = \left(M' \mathbin{;} M \,\middle|\, \phi\right)}$$

Figure 6.8: The operators $\boxed{\star}$ and $\boxed{\odot}$

$$\textsc{glue-frame-}\boxed{\star}\ \frac{\Phi, t \Downarrow \Phi'}{\phi \boxed{\star} \Phi, t \Downarrow \phi \boxed{\star} \Phi'}\ \mathit{itf}\left(\Phi\right) = \mathit{itf}\left(\Phi'\right) \qquad \textsc{glue-frame-}\boxed{\odot}\ \frac{\Phi, t \Downarrow \Phi'}{M \boxed{\odot} \Phi, t \Downarrow M \boxed{\odot} \Phi'}\ \mathit{itf}\left(\Phi\right) = \mathit{itf}\left(\Phi'\right)$$

Figure 6.9: The two framing rules

rules GLUE-FRAME-$\boxed{\star}$ and GLUE-FRAME-$\boxed{\odot}$, shown in Figure 6.9. These rules enforce the considered formulae to have the same interface. This is an invariant of all rules and is not a problematical constraint, although it can interfere with potential glue rules, as discussed in Section 6.4.5. They also enforce the two operators $\boxed{\star}$ and $\boxed{\odot}$ to be defined.

Consider the unsound abstract derivation of Figure 6.5. Figure 6.10 shows the corresponding derivation with membraned formulae. Membrane formulae are equipped with a renaming process: Rule GLUE-WEAKEN-$\preccurlyeq$ enables to replace an abstract location l_1 into an abstract location l_2, updating the membrane accordingly. This process is detailed in the next section. Because the membrane is also updated, the locations of the context formula added by the frame rule are also updated when getting in the membraned formula through the $\boxed{\star}$ operator. In a way, the rewriting of l_1 to l_2 has been stored in the membrane and propagated when Rule GLUE-FRAME-$\boxed{\star}$ was applied.

$$
\begin{array}{c}
\dfrac{\dfrac{\dfrac{}{(\eta_o \to \eta_i, l_0 \to l_1 \mid l_1 \mapsto \{\mathtt{f} : l_1\}, \eta_i, \eta_i), skip \Downarrow (\eta_o \to \eta_i, l_0 \to l_1 \mid l_1 \mapsto \{\mathtt{f} : l_1\}, \eta_i, \eta_i)}\ \textsc{red-skip}}{(\eta_o \to \eta_i, l_0 \to l_1 \mid l_1 \mapsto \{\mathtt{f} : l_1\}, \eta_i, \eta_i), skip \Downarrow (\eta_o \to \eta_i, l_0 \to l_2 \mid l_2 \mapsto \{\mathtt{f} : l_2\}, \eta_i, \eta_i)}\ \textsc{glue-weaken-}\preccurlyeq}{l_0 \mapsto \{\mathtt{g} : l_0\} \circledast (\eta_o \to \eta_i, l_0 \to l_1 \mid l_1 \mapsto \{\mathtt{f} : l_1\}, \eta_i, \eta_i), skip \Downarrow l_0 \mapsto \{\mathtt{g} : l_0\} \circledast (\eta_o \to \eta_i, l_0 \to l_2 \mid l_2 \mapsto \{\mathtt{f} : l_2\}, \eta_i, \eta_i)}\ \textsc{glue-frame-}\circledast \\
\overset{\star}{=} \qquad\qquad\qquad\qquad \overset{\star}{=} \\
(\eta_o \to \eta_i, l_0 \to l_1 \mid l_1 \mapsto \{\mathtt{g} : l_1,\ \mathtt{f} : l_1\}, \eta_i, \eta_i), skip \Downarrow (\eta_o \to \eta_i, l_0 \to l_2 \mid l_2 \mapsto \{\mathtt{g} : l_2,\ \mathtt{f} : l_2\}, \eta_i, \eta_i)
\end{array}
$$

Figure 6.10: A derivation showing how membranes protect renamed locations

$$
\begin{array}{c}
\dfrac{\dfrac{}{(\eta \to \eta_i \mid emp, \eta_i, \eta_i), alloc \Downarrow (\eta \to \eta_i, \bullet \to l \mid l \mapsto \{\_ : \boxtimes\}, \eta_i, l)}\ \textsc{red-new-obj}}{\big(l \to l', \eta_o \to \eta\big) \circledcirc (\eta \to \eta_i \mid emp, \eta_i, \eta_i), alloc \Downarrow \big(l \to l', \eta_o \to \eta\big) \circledcirc (\eta \to \eta_i, \bullet \to l \mid l \mapsto \{\_ : \boxtimes\}, \eta_i, l)}\ \textsc{glue-frame-}\circledcirc \\
= \qquad\qquad\qquad\qquad = \\
\dfrac{\big(l \to l', \eta_o \to \eta_i \mid emp, \eta_i, \eta_i\big), alloc \Downarrow \big(l \to l', \eta_o \to \eta_i, \bullet \to l \mid l \mapsto \{\_ : \boxtimes\}, \eta_i, l\big)}{l \mapsto \{\mathtt{f} : l\} \circledast \big(l \to l', \eta_o \to \eta_i \mid emp, \eta_i, \eta_i\big), alloc \Downarrow l \mapsto \{\mathtt{f} : l\} \circledast \big(l \to l', \eta_o \to \eta_i, \bullet \to l \mid l \mapsto \{\_ : \boxtimes\}, \eta_i, l\big)}\ \textsc{glue-frame-}\circledast \\
= \qquad\qquad\qquad\qquad = \\
\big(l \to l', \eta_o \to \eta_i \mid l' \mapsto \{\mathtt{f} : l'\}, \eta_i, \eta_i\big), alloc \Downarrow \big(l \to l', \eta_o \to \eta_i, \bullet \to l \mid l' \mapsto \{\mathtt{f} : l'\} \star l \mapsto \{\_ : \boxtimes\}, \eta_i, l\big)
\end{array}
$$

Figure 6.11: A derivation showing how membranes protect allocated locations

$$
\textsc{$\preccurlyeq$-refl}\ \dfrac{}{\Phi \preccurlyeq \Phi}
\qquad
\textsc{$\preccurlyeq$-trans}\ \dfrac{\Phi_1 \preccurlyeq \Phi_2 \qquad \Phi_2 \preccurlyeq \Phi_3}{\Phi_1 \preccurlyeq \Phi_3}
\qquad
\preccurlyeq\text{-}\circledast\ \dfrac{\Phi \preccurlyeq \Phi'}{\phi \circledast \Phi \preccurlyeq \phi \circledast \Phi'}
\qquad
\preccurlyeq\text{-}\circledcirc\ \dfrac{\Phi \preccurlyeq \Phi'}{M \circledcirc \Phi \preccurlyeq M \circledcirc \Phi'}
$$

$$
\textsc{$\preccurlyeq$-rename-obj}\ \dfrac{}{(l_0^\bullet \to l_1 \mid emp) \preccurlyeq (l_0^\bullet \to l_2 \mid emp)}
\qquad
\textsc{$\preccurlyeq$-rename-env}\ \dfrac{}{(\eta_0^\bullet \to \eta_1 \mid emp) \preccurlyeq (\eta_0^\bullet \to \eta_2 \mid emp)}
$$

$$
\textsc{$\preccurlyeq$-weaken-obj}\ \dfrac{u_1^\sharp \sqsubseteq u_2^\sharp}{\big(M \mid l \mapsto \{\mathtt{f} : u_1^\sharp\}\big) \preccurlyeq \big(M \mid l \mapsto \{\mathtt{f} : u_2^\sharp\}\big)}
\qquad
\textsc{$\preccurlyeq$-weaken-obj-cofinite}\ \dfrac{u_1^\sharp \sqsubseteq u_2^\sharp}{\big(M \mid l \mapsto \{\overline{F} : u_1^\sharp\}\big) \preccurlyeq \big(M \mid l \mapsto \{\overline{F} : u_2^\sharp\}\big)}
$$

$$
\textsc{$\preccurlyeq$-weaken-env}\ \dfrac{u_1^\sharp \sqsubseteq u_2^\sharp}{\big(M \mid \eta \mapsto \{\mathsf{x} \mapsto u_1^\sharp\}\big) \preccurlyeq \big(M \mid \eta \mapsto \{\mathsf{x} \mapsto u_2^\sharp\}\big)}
\qquad
\textsc{$\preccurlyeq$-weaken-ext}\ \dfrac{x_1^\sharp \sqsubseteq x_2^\sharp}{\big(M \mid emp, x_1^\sharp\big) \preccurlyeq \big(M \mid emp, x_2^\sharp\big)}
$$

Figure 6.12: Rewriting rules defining the operator $\preccurlyeq$

$$\dfrac{\dfrac{}{(l_0 \to l_1 \mid emp) \preccurlyeq (l_0 \to l_2 \mid emp)}\ \preccurlyeq\text{-}\textsc{rename-obj}}{l_0 \mapsto \{\mathtt{f} : l_0\} \boxed{\star} (l_0 \to l_1 \mid emp) \preccurlyeq l_0 \mapsto \{\mathtt{f} : l_0\} \boxed{\star} (l_0 \to l_2 \mid emp)}\ \preccurlyeq\text{-}\boxed{\star}$$

$$\stackrel{\star}{\doteq} \qquad\qquad \stackrel{\star}{\doteq}$$

$$(l_0 \to l_1 \mid l_1 \mapsto \{\mathtt{f} : l_1\}) \preccurlyeq (l_0 \to l_2 \mid l_2 \mapsto \{\mathtt{f} : l_2\})$$

Figure 6.13: Renaming a location using the rules of Figure 6.12

We now consider the issue of allocated locations presented in Figure 6.6. Figure 6.11 shows how membraned formulae protect such locations. Allocations introduce the use of $\bullet$ in membranes. In the derivation of Figure 6.11, Rule RED-NEW-OBJ picks a new abstract location l and states that it is newly allocated (and thus associated with $\bullet$) in the membrane. Let us try to frame the result with the formula $\phi = l \mapsto \{\mathtt{f} : l\}$, which uses the name l to represent a different location. It is not possible to frame it directly as $\mathit{itf}\,(\phi) = \{l\}$, but $\mathit{Out}\,(\eta_o \to \eta_i, \bullet \to l) = \{\eta_o\}$: the operator $\boxed{\star}$ is not defined and Rule GLUE-FRAME-$\boxed{\star}$ does not apply. To frame ϕ, we first have to extend the membrane so that l appears in its outer interface using Rule GLUE-FRAME-$\boxed{\odot}$. As for the previous example, the membrane made sure that the inner l and the outer l never mix. The intermediary membrane $M = (l \to l', \eta_o \to \eta_i, \bullet \to l)$ may be counter-intuitive as $\mathit{In}\,(M) \cap \mathit{Out}\,(M) = \{l\} \neq \varnothing$, but l represents different locations inside and outside the membrane. To avoid the confusion between the two abstract locations named l, it is possible to rename the inner l using Rule GLUE-WEAKEN-$\preccurlyeq$ as before, and only then apply the frame rule.

6.4.3 Rewriting Under Membraned Formulae

As we have seen in the previous section, membranes now scope locations, which can safely be renamed. We introduce a relation $\preccurlyeq$ over membraned formulae to perform these rewritings. It will be extended in Section 6.5.3 to enable approximating formulae. We incorporate $\preccurlyeq$ into our framework by the following glue rule.

$$\textsc{glue-weaken-}\preccurlyeq\quad \dfrac{\Phi_1 \preccurlyeq \Phi_2 \qquad \Phi_2, t \Downarrow \Phi_3 \qquad \Phi_3 \preccurlyeq \Phi_4}{\Phi_1, t \Downarrow \Phi_4}$$

The relation $\preccurlyeq$ is defined as the relation induced by the rules of Figure 6.12. The relation $\preccurlyeq$ is transitive and reflexive. Its rules introduce rewritings, then propagate them along the given formula. For instance Rule $\preccurlyeq$-RENAME-OBJ renames a location in a simple membrane, but does not perform any renaming on the attached formula, which is supposed to be *emp*. The renaming is propagated into the whole formula using Rule $\preccurlyeq$-$\boxed{\odot}$ to extend the interface of the formula, then Rule $\preccurlyeq$-$\boxed{\star}$ to let a formula ϕ pass through the membrane, thereby performing its renaming. The relation $\preccurlyeq$ also enables to weaken abstract values

$$\frac{\text{RED-LAMBDA}(\mathtt{x}, s)}{\left(- \mid emp, \eta_e, \eta_c\right), \lambda\mathtt{x}.s \Downarrow \left(- \mid emp, \eta_e, \left(\eta_c, \lambda\mathtt{x}.s\right)\right)}$$

$$\frac{\text{RED-APP-2}(s) \quad \eta_c' \text{ fresh} \qquad \left(-, \bullet \to \eta_c' \mid \eta_c' \mapsto E^\sharp\left[\mathtt{x} \leftarrow v^\sharp\right] \star \eta \mapsto E^\sharp \star \phi, \eta_e, \eta_c'\right), s \Downarrow \Phi \qquad \Phi, @_3 \Downarrow \Phi'}{\left(- \mid \eta \mapsto E^\sharp \star \phi, \eta_e, \eta_c, \mathtt{x}, s, v^\sharp\right), @_2 \Downarrow \Phi'}$$

$$\frac{\text{RED-NEW-OBJ}}{\left(- \mid emp, \eta_e, \eta_c\right), alloc \Downarrow \left(-, \bullet \to l \mid l \mapsto \{\_ : \boxtimes\}, \eta_e, l\right)}$$

$$\frac{\text{RED-FIELD-ASN-2}(\mathtt{f})}{\left(- \mid l \mapsto \left\{\mathtt{f} : u^\sharp\right\}, \eta_e, \eta_c, l, v^\sharp\right), .\mathtt{f} :=_2 \cdot \Downarrow \left(- \mid l \mapsto \left\{\mathtt{f} : v^\sharp\right\}, \eta_e, \eta_c\right)}$$

$$\frac{\text{RED-DELETE-1}(\mathtt{f})}{\left(- \mid l \mapsto \left\{\mathtt{f} : u^\sharp\right\}, \eta_c, \eta_e, l\right), delete_1\ .\mathtt{f} \Downarrow \left(- \mid l \mapsto \left\{\mathtt{f} : \boxtimes\right\}, \eta_c, \eta_e\right)}$$

Figure 6.14: A selection of abstract rules

through Rules ≼-WEAKEN-OBJ, ≼-WEAKEN-ENV, and ≼-WEAKEN-OBJ-COFINITE, as well as the potential extension of formulae through Rule ≼-WEAKEN-EXT. Figure 6.13 shows how to derive $\Phi_1 \preccurlyeq \Phi_2$, where Φ_1 and Φ_2 are the two formulae of the example of Figure 6.10.

We now present the main property of the $\preccurlyeq$ operator. This theorem states that when rewriting a membraned formula Φ_1 into a membraned formula Φ_2 using the $\preccurlyeq$ operator, then both the interface and the concretisation of the two formula are left unchanged. This property is also conserved when $\preccurlyeq$ is extended in Section 6.5.3.

Theorem 6.4. *For all Φ_1 and Φ_2 such that $\Phi_1 \preccurlyeq \Phi_2$, we have $itf\left(\Phi_1\right) = itf\left(\Phi_2\right)$. Furthermore, for all ρ, H_e, D, and H such that $\left(H_e, D, H\right) \vDash_\rho \Phi_1$, then $\left(H_e, D, H\right) \vDash_\rho \Phi_2$.*

6.4.4 Abstract Rules

The frame rule enables to only partly specify abstract rules. In particular, abstract rules can be defined on the resources which they need, without additional noise. For instance it is not an issue to only define Rule RED-NEW-OBJ on an empty formula, as we can extend the context using frame rules, as in Figure 6.11. This is a common practise in separation logic. For instance, the inference rule for variable in the work of Gardner, Maffeis, and Smith [GMS12] considers that the heap only contains a prototype chain to the looked-up variable. Such restrained abstract rules are simpler to define and to prove sound, thus reducing the effort needed to prove the soundness of the abstract semantics in Coq.

RED-LAMBDA$(\mathtt{x}, s)$

$$\frac{}{H, H_e, \ell_e, \ell_c, \lambda \mathtt{x}.s \Downarrow H, H_e, \ell_e, (\ell_c, \lambda \mathtt{x}.s)}$$

RED-APP-2(s)

$$\frac{\ell_c' = \mathit{fresh}\,(H_e) \qquad C = H_e\,[\ell_c] \qquad H, H_e\left[\ell_c' \leftarrow C\,[\mathtt{x} \leftarrow v]\right], \ell_e, \ell_c', s \Downarrow r \qquad r, @_3 \Downarrow r'}{\ell_c, \mathtt{x}, s, (H, H_e, \ell_e, v), @_2 \Downarrow r'}$$

RED-NEW-OBJ

$$\frac{\ell = \mathit{fresh}\,(H)}{H, H_e, \ell_e, \ell_c, \mathit{alloc} \Downarrow H\,[\ell \leftarrow \{\}], H_e, \ell_e, \ell}$$

RED-FIELD-ASN-2$(\mathtt{f})$

$$\frac{o = H\,[\ell] \qquad H' = H\,[\ell \leftarrow o\,[\mathtt{f} \leftarrow v]]}{\ell_c, \ell, (H, H_e, \ell_e, v), \mathtt{.f} :=_2 \cdot \Downarrow H', H_e, \ell_e, \ell_c} \quad \ell \in \mathit{dom}\,(H)$$

RED-DELETE-1$(\mathtt{f})$

$$\frac{o = H\,[\ell] \qquad H' = H\,[\ell \leftarrow o \setminus \mathtt{f}]}{\ell_c, (H, H_e, \ell_e, \ell), \mathit{delete}_1\ \mathtt{.f} \Downarrow H', H_e, \ell_e, \ell_c} \quad \ell \in \mathit{dom}\,(H)$$

Figure 6.15: The concrete rules corresponding to the abstract rules of Figure 6.14

As presented in the last two chapters, we abstract each rule independently, without considering the interaction between different rules. Figure 6.14 shows some abstract rules for our O'WHILE language. The corresponding concrete rules are shown in Figure 6.15. The other abstract rules can be found in the companion website [Bod16]. This section aims at showing that abstracting the concrete rules is a relatively straightforward process.

Consider the concrete Rule RED-LAMBDA$(\mathtt{x}, s)$ (repeated in Figure 6.15). Its semantic context (H, H_e, ℓ_e, ℓ_c) features the heap, the environment heap, and the two environment locations for the global and local environments (see Section 4.7.1). One may expect the semantic context of the corresponding abstract rule to be of the general form $(M \mid \phi, \eta_e, \eta_c)$. However, the concrete rule does not change or read the heaps H and H_e: they are not needed as resources for the rule and can be omitted in the abstract rule. The glue rules GLUE-FRAME-$\boxed{\star}$ and GLUE-FRAME-$\boxed{\odot}$ can be used at each application to remove the superfluous context, similarly to what is shown in Figure 6.11. The membrane of the abstract Rule RED-LAMBDA$(\mathtt{x}, s)$ is forced to be trivial: it is either of the form $(\eta_e \to \eta_e, \eta_c \to \eta_c)$ or $(\eta \to \eta)$, depending on whether the abstract environment locations η_e and η_c are identical. Indeed, the glue Rule GLUE-FRAME-$\boxed{\odot}$ can adapt such a trivial membrane to any other membrane performing complex rewritings. To simplify notations, we write trivial membranes, which are membranes only containing rewritings of the form $p \to p$, as $-$.

We have already described Rule RED-NEW-OBJ in Section 6.4.2. Rule RED-FIELD-ASN-2(s) is similar in the sense that it allocates a new environment identifier η_c', thus changing both the membrane and the formula. The action performed by the concrete rule is conveniently abstracted by separation logic: it only adds a new term in the state formula. Contrary to Rule RED-NEW-OBJ, Rule RED-FIELD-ASN-2(s) is a format 2 rule (see Section 4.3.1). In our formalism, the glue Rule GLUE-FRAME-$\boxed{\star}$ can only be used to extend already computed triples, but not derivations: it applies horizontally, but not vertically. To build a derivation from Rule red-field-asn-2(s), we thus have to provide its premises with their needed resources. As a consequence, we need to carry a formula ϕ in the semantic context.

Rule red-field-asn-2(`f`) writes the value $v^\sharp$ in the heap formula at computed location l. As it is an axiom, it only features the resources needed to apply the rule, which is the field `f` pointed by l. The rest of the object pointed by l can then be added by Rule glue-frame-$\boxed{\star}$. This rule assumes that it is provided exactly one abstract location l. This may not be the case: for instance, the abstract Rule red-field-asn-1$(\mathtt{f}, e_2)$ may result in a value of the form $l_1 \sqcup l_2 \sqcup b^\sharp$. This situation is covered by Rule glue-trace-partitioning of Section 5.1.2: this rule enables to cut such values and consider each case independently. Finally, the abstract Rule red-delete-1(`f`) behaves exactly the same than the abstract Rule red-field-asn-2(`f`): the only difference is that it sets the abstract field `f` to ⊠ instead of a computed abstract value $v^\sharp$.

6.4.5 Correctness of the Frame Rules

We have shown the definitions of the two frame Rules glue-frame-$\boxed{\star}$ and glue-frame-$\boxed{\star}$ in Figure 6.9. To use these rules, we have to prove them correct. We have tried to prove the correctness of these rules with respect to Criterion 5.3 of Section 5.2.2. This criterion is based on concrete derivation rewritings. Derivation rewritings fit by design the requirements of separation logic, as illustrated in Figure 5.5. Despite our efforts, it proved to be more challenging than expected and is currently admitted in our Coq development.

In a nutshell, the issue with Criterion 5.3 is that it crucially depends on the concretisation γ. In particular, it enables glue rules such as Rule glue-weaken to change membranes—and thus changing the global meaning of locations—as long as the concretisation does not change. More precisely, we proved that Criterion 5.3 does not hold as-is for the two frame Rules glue-frame-$\boxed{\star}$ and glue-frame-$\boxed{\star}$. A detailed proof is given in the website accompanying this dissertation [Bod16]. This does not mean that these two rules are unsound, only that our correctness criterion is not enough to prove their soundness.

To prove the soundness of these glue rules, we need to update Criterion 5.3 to prevent Rule glue-weaken from modifying membranes carelessly. In particular, we would like to force this rule to consider some global invariants used by separation logic. The issue is that the current concretisation function for formulae (repeated below without formula extension for simplicity—see Equation 6.8 for the full definition) does not make explicit the choice of the valuation ρ_o from the outside: if we know that $(H_e, H) \in \gamma(\Phi)$, we have no idea which valuation ρ_o has been used to instantiate the concretisation relation.

$$(H_e, H) \in \gamma((M \mid \phi)) \iff \exists \rho_i, \rho_o, \nu.\ (\rho_o, \nu, \rho_i) \in \gamma(M) \wedge (H_e, H) \vDash_{\rho_i} \phi$$

To make this valuation choice explicit, we propose an alternative parametrised definition for the concretisation of membraned formulae γ_{ρ_o}, shown below.

$$(H_e, H) \in \gamma_{\rho_o}((M \mid \phi)) \iff \exists \rho_i, \nu.\ (\rho_o, \nu, \rho_i) \in \gamma(M) \wedge (H_e, H) \vDash_{\rho_i} \phi$$

This new definition makes explicit which valuation ρ_o has been used in the concretisation function. We can now update the definition of the k correctness used in Criterion 5.3 as follows: a semantic triple $\sigma^\sharp, t \Downarrow^\sharp r^\sharp$ is k correct, if for any concrete derivation of depth less than k with conclusion $\sigma, t \Downarrow r$, and any valuation ρ, then $\sigma \in \gamma_\rho(\sigma^\sharp)$ implies $r \in \gamma_\rho(r^\sharp)$. This new definition of the k correctness forces the invariant needed by separation logic—that locations do not change their concrete representation along derivations—to be propagated. In particular, Rule GLUE-WEAKEN is no longer allowed to carelessly modify membranes: it can still modify the inner interface $In(M)$ of a membrane M, but not its outer interface $Out(M)$ as this would invalidate the outer valuation ρ_o. It is still possible for Rule GLUE-WEAKEN to weaken an abstract value, which is compatible with Rules GLUE-FRAME-$\boxed{\star}$ and GLUE-FRAME-$\boxed{\star}$. The behaviours of Rule GLUE-WEAKEN likely to be use in an abstract derivation are in fact those of the $\preccurlyeq$ operator, shown in Figure 6.12.

As this is on-going work, we do not change the presentation of the formalism in the rest of this dissertation. Although incomplete, we have good hope in proving that our logic is sound in our Coq formalisation.

6.5 Shapes and Summary Nodes

As-is, our domains are not adequate for recursive data structures, whose size may vary. To address this issue, we extend our certified abstract semantics with abstractions coming from shape analysis [SRW98], in particular the notion of summary nodes. The amount of effort needed to extend our formalism revealed itself to be surprisingly small. Membranes indeed naturally express the summary operations of shape analyses. We start by showing that membranes as they currently are already enable to introduce some approximations.

6.5.1 Abstracting Using Membranes

Membranes prevent identifier collisions when using the frame rules, but they can also be used to express approximations. This is hinted by the fact that the relation $\preccurlyeq$ renaming identifiers is a preorder relation and not an equivalence relation (it is not antisymmetrical): some membranes are more precise than others. We now explore this aspect of membranes. We now allow membranes to be non-functional and non-injective: we now write them as a list of relations of the form $p^\bullet \to p_1 + \ldots + p_n$, collecting all the locations p_i related to $p^\bullet$. In particular, an outer location can be related to several inner locations—it then intuitively corresponds to one of these inner locations.

Consider the membrane $M_r = (l_1 \to l'_1 + l'_2, l_2 \to l'_1 + l'_2)$. Each of the outer locations l_1 and l_2 may be mapped into one of the inner locations l'_1 and l'_2, and, conversely, each of l'_1 and l'_2 may be the image of l_1 or l_2. The membrane M_r has lost some information: we know that the set of outer locations $\{l_1, l_2\}$ represents the same concrete locations than the set of inner locations $\{l'_1, l'_2\}$, but we have lost the exact relation between locations.

In order to be well formed, membraned formulae $(M \mid \phi)$ must satisfy an additional requirement: related inner locations in $dom\,(\phi)$ must map to objects or environments œ with the same specified domain *spec*. For instance, we impose $spec\left(o_2^\sharp\right) = spec\left(o_3^\sharp\right)$ on the membraned formula $\left(l_1 \to l_2 + l_3 \mid l_2 \mapsto o_2^\sharp \star l_3 \mapsto o_3^\sharp\right)$. We impose this restriction to be able to define the specified domain of outer locations: every object or environment associated to an outer location musts specify the same domain. Note that this is trivial for environments as their specified domain *spec* is always Var.

$$\forall p_o, p_i, p'_i.\ M\,(p_o, p_i) \wedge M\left(p_o, p'_i\right) \implies \bigcup_{p_i \to \text{œ}_i^\sharp \in \phi} spec\left(\text{œ}_i^\sharp\right) = \bigcup_{p'_i \to \text{œ}'^\sharp_i \in \phi} spec\left(\text{œ}'^\sharp_i\right)$$

As a consequence, the operator $\triangleright$ for object crossing membrane has to check that all the needed information is given. For instance, consider the membrane M_r defined above. The formula $l_1 \mapsto o_1^\sharp \triangleright M_r$ is undefined as the corresponding information about l_2 is missing. Once this information given, we can write $\left(l_1 \mapsto o_1^\sharp \star l_2 \mapsto o_2^\sharp \triangleright M_r\right) = l'_1 \mapsto o'^\sharp \star l'_2 \mapsto o'^\sharp$, where $o'^\sharp = \left(o_1^\sharp \triangleright M_r\right) \sqcup \left(o_2^\sharp \triangleright M_r\right)$, if $spec\left(o_1^\sharp\right) = spec\left(o_2^\sharp\right)$. We recall from Section 6.3.3 that objects with the same specified domain form a lattice, hence the $\sqcup$ operator. This process is generalised by the complex fourth rule of Figure 6.7. This rule ensures that all objects with common dependencies cross the membrane at the same time. The condition $\dot{M}^{-1}\left(\dot{M}\,(\Lambda)\right) = \Lambda$ ensures that Λ does not miss any needed inner location.

6.5.2 Summary Nodes and Membranes

So far, concrete locations ℓ have been abstracted by abstract locations $l \in L$, each of them representing exactly one concrete location. We now introduce summary nodes $k \in K$, which can abstract any set of concrete locations. We update abstract values to include summary nodes: we now have $v^\sharp \in Val^\sharp = Sign^\sharp \times \mathcal{P}\,(L \uplus K) \times \mathcal{C}^\sharp$. The definition of formulae is adapted accordingly. We write $h \in L \uplus K$ for an abstract location—which can be either a simple abstract location l or a summary node k—, and Λ for a subset of $L \uplus K$. The interface $itf\,(\phi) \subseteq L \uplus K \uplus \mathcal{L}_e^\sharp$ and the domain $dom\,(\phi) \subseteq ((L \uplus K) \times Field) \uplus \left(\mathcal{L}_e^\sharp \times Var\right)$ of a formula ϕ is updated as expected.

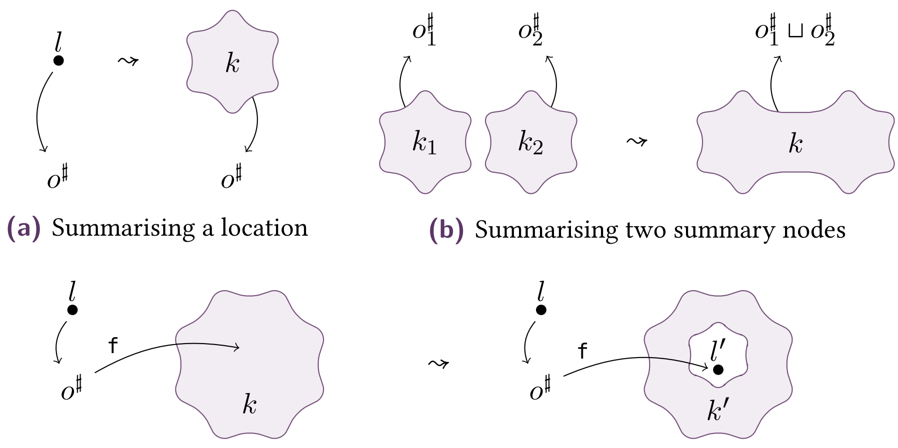
$$\phi ::= \mathit{emp} \mid \phi_1 \star \phi_2 \mid \eta \mapsto env^\sharp \mid h \mapsto o \qquad\qquad h ::= l \mid k$$

The purpose of summary nodes is to enable the merging of locations. For instance several abstract locations $l_1, \ldots, l_n$ can be merged into a single summary node k to simplify the formula. Such an approximation changes the interface of formulae and can be reflected in a

$$\frac{\mathit{dom}\,(H) = \rho\,(h) \qquad \forall \ell \in \rho\,(h)\,.\; H\,[\ell] \in \gamma_\rho\left(o^\sharp\right)}{\left(\epsilon, \mathit{dom}\,(H) \times \mathit{spec}\left(o^\sharp\right), H\right) \vDash_\rho h \mapsto o^\sharp}$$

$$\frac{\begin{array}{c}\mathit{dom}\,(E) = \rho\,(\eta) \qquad \forall \ell_e \in \rho\,(\eta)\,.\; \mathit{dom}\,(E\,[\ell_e]) = \mathit{dom}\left(E^\sharp\right) \\ \forall \ell_e \in \rho\,(\eta)\,, \mathsf{x} \in \mathit{dom}\left(E^\sharp\right).\; E\,[\ell_e]\,[\mathsf{x}] \in \gamma_\rho\left(E^\sharp\,[\mathsf{x}]\right)\end{array}}{(E, \varnothing, \epsilon) \vDash_\rho \eta \mapsto E^\sharp}$$

Figure 6.16: Updating the last two rules of Figure 6.4 for the entailment



(a) Summarising a location

(b) Summarising two summary nodes

(c) Materialisation

Figure 6.17: Visualisation of membrane operations

membrane. For instance, merging l_1 and l_2 into k results in the membrane $(l_1 \rightarrow k, l_2 \rightarrow k)$. For simplicity, environment identifiers η are considered as summary nodes (their concretisation is a set of environment locations). We could have introduced the same granularity for environment identifiers, but it is not needed to demonstrate the approach. To accommodate summary nodes, we redefine membranes as

$$M \in \mathcal{P}_{\mathit{fin}}\,((L \uplus K \uplus \{\bullet\}) \times (L \uplus K)) \uplus \mathcal{P}_{\mathit{fin}}\left(\left(\mathcal{L}_e^\sharp \uplus \{\bullet\}\right) \times \mathcal{L}_e^\sharp\right)$$

The definitions of inner and outer interfaces, membraned formulae, membrane composition $⨟$, and membrane crossing $\triangleright$ remain unchanged. Valuations ρ map l to singleton sets $\{\ell\}$, k to sets of locations, and η to sets of environment locations. The entailment predicate (see Figure 6.4) is modified as shown in Figure 6.16.

6.5.3 Approximation Rules With Summary Nodes

We now have the tools to define rules for approximating membraned formulae. The presented approximations are informally depicted in Figure 6.17; these approximations are usual in shape analysis. We complete the order $\preccurlyeq$ (see Figure 6.12) by the rules of Figure 6.18. As

$$\frac{}{(l_o \to l_i \mid emp) \preccurlyeq (l_o \to k_i \mid emp)}\ \textsc{$\preccurlyeq$-Loc} \qquad \frac{}{\left(h_o \to h_i, h_o' \to h_i' \mid emp\right) \preccurlyeq \left(h_o \to k_i, h_o' \to k_i \mid emp\right)}\ \textsc{$\preccurlyeq$-Merge}$$

$$\frac{}{(\epsilon \mid emp) \preccurlyeq \left(\bullet \to k \mid k \mapsto o^\sharp\right)}\ \textsc{$\preccurlyeq$-Void} \qquad \frac{}{(l_o \to l_i, k_o \to k_i \mid l_i \to \{\mathtt{f} : k_i\}) \preccurlyeq \left(l_o \to l_i, k_o \to k_i' + l_i' \mid l_i \to \{\mathtt{f} : l_i'\}\right)}\ \textsc{$\preccurlyeq$-Materialise}$$

$$\frac{}{(l \to k \mid emp) \preccurlyeq \left(l \to l' \mid emp\right)}\ \textsc{$\preccurlyeq$-Materialise-Membrane} \qquad \frac{}{(k_o \to k_i \mid emp, k_i) \preccurlyeq (k_o \to l_i \mid emp, l_i)}\ \textsc{$\preccurlyeq$-Materialise-Ext}$$

$$\frac{h \in In\,(M)}{(M, h_0 \to h_1 + \ldots + h_n \mid emp) \preccurlyeq (M, h_0 \to h + h_1 + \ldots + h_n \mid emp)}\ \textsc{$\preccurlyeq$-Extend}$$

$$\frac{}{\left(\eta_o \to \eta_i, \eta_o' \to \eta_i' \mid emp\right) \preccurlyeq \left(\eta_o \to \eta_i'', \eta_o' \to \eta_i'' \mid emp\right)}\ \textsc{$\preccurlyeq$-Merge-Env} \qquad \frac{}{(\epsilon \mid emp) \preccurlyeq \left(\bullet \to \eta \mid \eta \mapsto E^\sharp\right)}\ \textsc{$\preccurlyeq$-Void-Env}$$

$$\frac{\eta \in In\,(M)}{(M, \eta_0 \to \eta_1 + \ldots + \eta_n \mid emp) \preccurlyeq (M, \eta + \eta_0 \to \eta_1 + \ldots + \eta_n \mid emp)}\ \textsc{$\preccurlyeq$-Extend-Env}$$

Figure 6.18: Rules for introducing approximations

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\left(- \mid l_i' \mapsto \left\{\mathtt{f} : u^\sharp\right\}, l_i', v^\sharp, \eta_e, \eta_c\right), .\mathtt{f} :=_2 \cdot \Downarrow \left(- \mid l_i' \mapsto \left\{\mathtt{f} : v^\sharp\right\}, \eta_e\right)}\ \textsc{red-field-asn-2}(\mathtt{f})}{\left(- \mid l_i' \mapsto \left\{\mathtt{f} : u^\sharp\right\} \star k_i' \mapsto \left\{\mathtt{f} : u^\sharp\right\}, l_i', v^\sharp, \eta_e, \eta_c\right), .\mathtt{f} :=_2 \cdot \Downarrow \left(- \mid l_i' \mapsto \left\{\mathtt{f} : v^\sharp\right\} \star k_i' \mapsto \left\{\mathtt{f} : u^\sharp\right\}, \eta_e\right)}\ \textsc{glue-frame-}\boxdot}{\left(-, k_o \to l_i' + k_i' \mid l_i' \mapsto \left\{\mathtt{f} : u^\sharp\right\} \star k_i' \mapsto \left\{\mathtt{f} : u^\sharp\right\}, l_i', v^\sharp, \eta_e, \eta_c\right), .\mathtt{f} :=_2 \cdot \Downarrow \left(-, k_o \to l_i' + k_i' \mid l_i' \mapsto \left\{\mathtt{f} : v^\sharp\right\} \star k_i' \mapsto \left\{\mathtt{f} : u^\sharp\right\}, \eta_e\right)}\ \textsc{glue-frame-}\boxdot}{\left(-, k_o \to k_i \mid k_i \mapsto \left\{\mathtt{f} : u^\sharp\right\}, k, v^\sharp, \eta_e, \eta_c\right), .\mathtt{f} :=_2 \cdot \Downarrow \left(-, k_o \to k_i \mid k \mapsto \left\{\mathtt{f} : u^\sharp \sqcup v^\sharp\right\}, \eta_e\right)}\ \textsc{glue-weaken-}\preccurlyeq$$

Figure 6.19: A weak update derived from a strong update

in Section 6.4.3, the rules only introduce membranes on simple formulae, then Rules $\preccurlyeq$-$\boxdot$ and $\preccurlyeq$-$\boxast$ complete the missing context. Rule $\preccurlyeq$-Loc is the basic approximation rule for replacing an inner location l_i with a summary node k_i in a membrane, changing a membrane of the form $(l_o \to l_i)$ to the membrane $(l_o \to k_i)$. Rules $\preccurlyeq$-Merge and $\preccurlyeq$-Merge-Env describe how two summary nodes can be merged into one, as pictured in Figure 6.17b.

Rule $\preccurlyeq$-Void describes how a summary node can be introduced from no location. There is indeed no constraint enforcing the valuation of a summary node to result in a non-empty set. We can thus make summary nodes appear out of thin air: the associated valuation ρ associates an empty set to such a summary node. We can state any kind of properties for such void summary nodes. Rule $\preccurlyeq$-Materialise formalises the principle of materialisation, extracting a location from a summary node. It requires an entry point to a summary node k, that is, an object field whose value is exactly k, and whose host object o is referenced by a precise abstract location l. Such an entry point ensures that the valuation $\rho(k)$ of the summary node k is not empty. We can thus divide the concrete set $\rho(k)$ of location in two: the entry point location l' and the rest k' of the summary node. The next section provides an example of how this rule can be used. Rule $\preccurlyeq$-MATERIALISE-EXT is a similar rule, but accepts as entry point a formula extension: we know that in the concrete derivation, there will be a location ℓ at this place, and we can thus consider it.

Rule $\preccurlyeq$-EXTEND adds inner locations associated to an outer location. For instance, this rule rewrites the precise membrane $(l_1 \to l'_1, l_2 \to l'_2)$ into the less precise membrane $M_r = (l_1 \to l'_1 + l'_2, l_2 \to l'_1 + l'_2)$ encountered above. Figure 6.18 shows similar rules for environment locations η, which behave like summary nodes. Each of these rules are defined such that if $\left(M_1 \mid \phi_1, x^\sharp\right) \preccurlyeq \left(M_2 \mid \phi_2, x_2^\sharp\right)$, then for all concretisation $(\rho_o, \nu, \rho_i) \in \gamma(M_1)$, $(H_e, H) \vDash_{\rho_i} \phi_1$, and $x \in \gamma_{\rho_i}\left(x_1^\sharp\right)$, then there exists an inner valuation ρ'_i such that $(\rho_o, \nu, \rho'_i) \in \gamma(M_2)$, $(H_e, H) \vDash_{\rho'_i} \phi_2$, and $x \in \gamma_{\rho'_i}\left(x_2^\sharp\right)$: these rules are simple enough that only the inner valuation ρ_i needs to change when the membraned formula $\left(M_1 \mid \Phi_1, x_1^\sharp\right)$ is rewritten to $\left(M_2 \mid \Phi_2, x_2^\sharp\right)$. In particular, the equations 6.5 to 6.7 of Section 6.4.1 still apply. This makes the proof of soundness of the rules of Figure 6.18 simple to do.

Shape analyses usually differentiate between strong updates and weak updates. We have only specified strong updates through the abstract Rule RED-FIELD-ASN-2($\mathtt{f}$) of Figure 6.14: the value $u^\sharp$ stored in the object before the assignment has been completely removed and replaced by the new value $v^\sharp$. For summary nodes k, it is not sound to replace the old value as only one concrete location ℓ of the valuation $\rho(k)$ of the summary node has been updated: the old value may still be present in other locations. We thus usually use the following rule performing a weak update: it merges the new value with the old value.

$$\frac{\textsc{red-field-asn-2}(\mathtt{f})}{\left(- \mid k \mapsto \left\{\mathtt{f} : u^\sharp\right\}, \eta_e, \eta_c, k, v^\sharp\right), \mathtt{.f} :=_2 \cdot \Downarrow \left(- \mid k \mapsto \left\{\mathtt{f} : u^\sharp \sqcup v^\sharp\right\}, \eta_e, \eta_c\right)}$$

This rule is not in our abstract semantics as it is deductible from materialisations: Figure 6.19 shows such an derivation. The derivation proceeds in four steps, to be read clockwise from the initial semantic context below left to the final conclusion below right. First Rule GLUE-WEAKEN-$\preccurlyeq$ performs a materialisation using Rule $\preccurlyeq$-MATERIALISE-EXT to split the summary node k_i into l'_i and k'_i. To simplify, we consider that the abstract values $u^\sharp$ and $v^\sharp$ are not affected by this membrane transformation. Second, the context is framed to only focus on the location l'_i on which the strong update will be performed. In particular, the summary node k'_i has been removed from the formula. Third, the strong update is performed: the value $u^\sharp$ is replaced by $v^\sharp$. But the summary node k'_i has not been updated and its value is still $u^\sharp$. Fourth, the abstract locations l'_i and k'_i are merged back again in Rule GLUE-WEAKEN-$\preccurlyeq$ using Rule $\preccurlyeq$-MERGE. Note that although syntactically similar, the two weakenings performed by this rule are not using the same rules from Figure 6.18.

$$\dfrac{\dfrac{\vdots}{(M_i \mid \phi_i, \eta_i, \eta_c, \top_{\mathbb{Z}}), s \Downarrow (M_i' \mid \phi_i', \eta_i', \eta_c)} \qquad \dfrac{\dfrac{(M_i \mid \phi_i, \eta_i, \eta_c), \mathit{while}\ (\imath > 0)\ s \Downarrow \Phi}{(M_i' \mid \phi_i', \eta_i', \eta_c), \mathit{while}\ (\imath > 0)\ s \Downarrow \Phi}\ \textsc{glue-weaken-}\preccurlyeq}{(M_i' \mid \phi_i', \eta_i', \eta_c), \mathit{while}_2\ (\imath > 0)\ s \Downarrow \Phi}\ \textsc{red-while-1-pos}(\imath, s)}{(M_i, \bullet \to l_1 \mid \phi_1, \eta, \eta_c, \top_{\mathbb{Z}},), \mathit{while}_1\ (\imath > 0)\ s \Downarrow \Phi}\ \textsc{red-while-1-pos}(\imath, s)$$

$$\dfrac{\dfrac{}{(M_i \mid \phi_i, \eta_i, \eta_c, \top_{\mathbb{Z}}), \mathit{while}_1\ (\imath > 0)\ s \Downarrow (M_i \mid \phi_i, \eta_i)}\ \textsc{red-while-1-neg}(\imath, s)}{(M_i \mid \phi_i, \eta_i, \eta_c, \top_{\mathbb{Z}}), \mathit{while}_1\ (\imath > 0)\ s \Downarrow \Phi}\ \textsc{glue-weaken-}\preccurlyeq$$

$$\dfrac{\dfrac{\vdots}{(M_0 \mid \phi_0, \eta_0, \eta_c), \mathtt{x} := \mathit{alloc} \Downarrow (M_1 \mid \phi_1, \eta_1, \eta_c)} \qquad \dfrac{\dfrac{\dfrac{\textsc{red-var-global}(\imath)\ \dfrac{}{(M_i \mid \phi_i, \eta_i, \eta_c), \imath \Downarrow (M_i \mid \phi_i, \eta_i, \top_{\mathbb{Z}})} \qquad (M_i \mid \phi_i, \eta_i, \eta_c, \top_{\mathbb{Z}}), \mathit{while}_1\ (\imath > 0)\ s \Downarrow \Phi}{(M_i \mid \phi_i, \eta_i, \eta_c), \mathit{while}\ (\imath > 0)\ s \Downarrow \Phi}\ \textsc{red-while}(\imath, s)}{(M_1 \mid \phi_1, \eta_1, \eta_c), \mathit{while}\ (\imath > 0)\ s \Downarrow \Phi}\ \textsc{glue-weaken-}\preccurlyeq}{(M_1 \mid \phi_1, \eta_1, \eta_c), \cdot ;_1 \mathit{while}\ (\imath > 0)\ s \Downarrow \Phi}\ \textsc{red-seq-1}(\mathit{while}\ (\imath > 0)\ s)}{(M_0 \mid \phi_0, \eta_0, \eta_c), \mathtt{x} := \mathit{alloc}; \mathit{while}\ (\imath > 0)\ s \Downarrow \Phi}\ \textsc{red-seq}(\mathtt{x} := \mathit{alloc}, \mathit{while}\ (\imath > 0)\ s)$$

$$M_0 = (\eta \to \eta_0, \eta' \to \eta_c) \qquad \phi_0 = \eta_0 \mapsto \{\imath \mapsto \top_{\mathbb{Z}}\} \star \eta_c \mapsto \{\_ \mapsto \boxtimes\} \qquad M_1 = (\eta_o \to \eta_0, \eta' \to \eta_c, \bullet \to l_1 + \eta_1)$$

$$\phi_1 = \phi_0 \star \eta_1 \mapsto \{\imath \mapsto \top_{\mathbb{Z}}, \mathtt{x} \mapsto l_0\} \star l_0 \mapsto \{\_ : \boxtimes\} \qquad M_i = (\eta_o \to \eta_g, \eta' \to \eta_c, \bullet \to k + \eta_i + \eta_g)$$

$$\phi_i = \eta_g \mapsto \{\imath \mapsto \top, \mathtt{x} \mapsto \top, \mathtt{t} \mapsto \top\} \star \eta_c \mapsto \{\_ \mapsto \boxtimes\} \star \eta_i \mapsto \{\imath \mapsto \top, \mathtt{x} \mapsto k\} \star k \mapsto \{\mathtt{next} : k \sqcup \boxtimes, \_ : \boxtimes\}$$

$$M_i' = (\eta_o \to \eta_g, \eta' \to \eta_c, \bullet \to k + \eta_i' + \eta_g + l + \eta_f) \qquad \phi_i' = \eta_g \mapsto \{\imath \mapsto \top, \mathtt{x} \mapsto \top, \mathtt{t} \mapsto \top\} \star \eta_c \mapsto \{\_ \mapsto \boxtimes\} \star \eta_i' \mapsto$$

$$\{\imath \mapsto \top, \mathtt{x} \mapsto l\} \star \eta_f \mapsto \{\mathtt{t} \mapsto l\} \star k \mapsto \{\mathtt{next} : k \sqcup \boxtimes, \_ : \boxtimes\} \star l \mapsto \{\mathtt{next} : k, \_ : \boxtimes\}$$

Figure 6.20: Beginning of the abstract derivation of the example program

6.5.4 Example

Separation logic is often extended with precise structures depending on the analysed programs—usually inductive structures [BG14]. Our framework does not forbid the use of these structures, but we chose a less precise abstraction. Summary nodes, on the other hand, are a generic abstraction and do not depend on the precise structure used in the analysed programs. We illustrate our analysis through an example manipulating lists.

Consider the following program creating a linked list of size given by the variable $\imath$, whose precise value we do not know and abstract by $\top_{\mathbb{Z}}$. We use a non-pure function to modify a local object $\mathtt{t}$. This example is interesting as it makes use of the two rules allocating locations: Rules RED-NEW-OBJ and RED-APP-2(s).

$$\mathtt{x} := \mathit{alloc}; \mathit{while}\ (\imath > 0)\ \mathtt{x} := (\lambda \mathtt{t}.\mathtt{t}.\mathtt{next} := \mathtt{x}; \imath := \imath + (-1); \mathit{return}\ \mathtt{t})\ (\mathit{alloc})$$

Let s be the loop body $\mathtt{x} := (\lambda \mathtt{t}.\mathtt{t}.\mathtt{n} := \mathtt{x}; \mathtt{l} := \mathtt{l} + (-1)\,; \mathit{return}\ \mathtt{t})\,(\mathit{alloc})$. Figure 6.20 shows the beginning of the abstract derivation, on the semantic context $(M_0 \mid \phi_0, \eta_0) = (\eta_o \to \eta_0, \eta' \to \eta_c \mid \eta_0 \mapsto \{\mathtt{l} \mapsto \top_{\mathbb{Z}}\} \star \eta_c \mapsto \{\_ \mapsto \boxtimes\}\,, \eta_0, \eta_c)$. The resulting membraned formula Φ is still to be found. This example does not use the local context η_c, which is mapped to an empty environment. The derivation starts by evaluating the assignment $\mathtt{x} := \mathit{alloc}$. This allocates two new locations: a location l_1 for the new object, but also a location η_1 for the new environment allocated to store x. The created membrane M_1 thus contains a rewriting $\bullet \to l_1 + \eta_1$ allocating these two locations (see Figure 6.20).

As for the example of Section 4.4.2.2, it is not possible to determine in advance the number of steps in the execution of this program. The abstract derivation is thus infinite. We build this infinite derivation by exhibiting an invariant. Because of the allocations, membranes and formulae increase over the abstract derivation. To find an invariant, we have merged the allocated locations into a summary node k. In the example, all locations are merged into one, but smarter mergings could be performed if needed. We only need one environment at a time in this example, other environments being old versions of the current environment. There is however no way to garbage collect [MS06; Jag+98] environments in our semantics, both concrete and abstract. We thus merge all previously used environments into a garbage summary node η_g, whose associated environment associates all variables to $\top$. Note that the external initial environment location η_o has also been merged into this summary node: we can not remove it (as it would change the interface of the membraned formula), and it has to be linked with the associated inner location. The introduction of the invariant is performed by Rule GLUE-WEAKEN-$\preccurlyeq$.

When reaching the term $\mathit{while}_1\ (\mathtt{l} > 0)\ s$, both Rules RED-WHILE-1-NEG$(\mathtt{l}, s)$ and RED-WHILE-1-POS$(\mathtt{l}, s)$ apply. Both rules are thus considered in parallel, as explained in Section 4.4.2.1. The application of Rule RED-WHILE-1-NEG$(\mathtt{l}, s)$ stops the evaluation in the membraned formula $(M_i \mid \phi_i, \eta_i)$. To relax the constraints on the final result Φ, we first apply Rule GLUE-WEAKEN-$\preccurlyeq$: we have the constraint $(M_i \mid \phi_i, \eta_i) \preccurlyeq \Phi$ about the yet unknown Φ. During the execution of the loop body, several locations are created: a new location l is allocated for the object creation, an environment location η_f for the function call, as well as a new environment location η_i' for the update of the global environment, whose details we hide. The body results in the membraned formula $(M_i' \mid \phi_i', \eta_i')$. This is not exactly our invariant, but Rule GLUE-WEAKEN-$\preccurlyeq$ is applied again to approximate it to our resulting invariant. We could also have applied Rules GLUE-FRAME-$\boxed{\star}$ and GLUE-FRAME-$\boxed{\circledcirc}$ to remove the allocated environment locations. The top of the derivation ends with the already seen semantic triple $(M_i \mid \phi_i, \eta_i)\,, \mathit{while}\ (\mathtt{l} > 0)\ s \Downarrow \Phi$: it has thus been proven to be an invariant, and the infinite derivation loops back to the first occurrence of this semantic triple, just above the first application of Rule GLUE-WEAKEN-$\preccurlyeq$

Overall, the only constraint collected on the membraned formula Φ was $(M_i \mid \phi_i, \eta_i) \preccurlyeq \Phi$. We thus define Φ to be exactly $(M_i \mid \phi_i, \eta_i)$. The final result states that the variable x points to a set of concrete locations $\rho(k)$, each with exactly the field next, whose value is either a location to the same set $\rho(k)$, or is undefined. This is not exactly a description of a list: it may be a looping list, or there could be several lists in the set $\rho(k)$. It is however an interesting result. This example shows how our glue rules—and in particular Rule GLUE-WEAKEN-$\preccurlyeq$—can be used to build derivations. We believe that this formalism provides a useful framework for proving analysers sound. We have used summary nodes to abstract lists, but the summary node abstraction can be used for various data structures. In particular, any points-to analysis [JC09] could be expressed in this framework.

6.6 Related Work and Conclusion

Our formalisation shares some interesting traits with analyses targeting JavaScript. In particular the TAJS analyser of Jensen, Møller, and Thiemann [JMT09] and the work of Cox, Chang, and Rival [CCR14]. Both of these works aim at building real-world analysers for JavaScript. These works are not yet related to JSCert or λ_{JS} and could be a good target to apply our formalism in further work: our formalism aims at building certified abstract semantics, which can then be related to concrete analysers (see Section 3.1). This would probably require a significant amount of work, as there are non-negligible differences between these two projects and our formalism. For instance, the TAJS project relies on a bytecode for JavaScript and thus uses a very different structure than the pretty-big-step of JSCert. We now consider how they abstract the memory model of JavaScript to evaluate how practical our formalism would be on these two projects.

In both cases, objects are abstracted as finite maps from fields to abstract values with a default value—very similarly to our cofinite objects (see Section 6.3.3)—, but both formalisms come with their specificities. TAJS specifies two default values: one for array indexes and one for other fields. Array indexes are fields whose name is parsable as a positive integer. Program 1.4 of Section 1.2.6 provides an example of a situation in which such a separation can be crucial. Their abstract values are much more precise than the one presented in this chapter, but they fit the constraints which we imposed on basic values $b^\sharp$: they provide a complete lattice, and thus a poset. An interesting common point is the way store values are defined in TAJS, rewritten below with our notations.

$$u^\sharp \in \mathit{Store}^\sharp = \mathit{Val}^\sharp \times \{\square, \boxtimes\} \times \mathit{flag} \times \mathit{modified}$$

In particular, they also use the product poset of Definition 3.2. The set *flag* represents JavaScript flags such as writable or enumerable (see Section 2.5.1.2). The set *modified* is discussed in the next paragraph. The second formalism (of Cox et al.) features only one such default field, named `noti` in their formalisation. However, their formalism enables to perform summarisation of fields. This is an interesting feature: in our formalism, we only summarise locations—never the fields of an object (apart from the default value of cofinite objects). This feature has been introduced to deal with **`for-in`** constructs. Their formalism enables to consider a (symbolic) set F of fields, to constrain this set using pure formulae of separation logic, then specify that all the fields of F associated to a given object are abstracted by a given abstract value $u^\sharp$. The drawback of this formalisation choice is that formulae are much more complex, carrying constraints such as $F_1 \cap F_2 = \varnothing$ or $F_1 \subseteq F_2$. Abstract operations have to check whether they are sound with respect to all these constraints. Our formalism is not yet ready to deal with such complex constructs.

Our formalism features an interesting aspect: objects can be partial. For instance a function which only needs the fields `f` and `g` of a object—for example to read the value of `f` and update the value of `g`—may be specified by a partial specification. An example of such a partial specification could take as argument an abstract object of the form $\{\texttt{f} : +, \texttt{g} : \top\}$ and return an abstract object of the form $\{\texttt{f} : +, \texttt{g} : +\}$. When given a fully specified object, it will be split into the needed part and the untouched parts. For instance the object $\{\texttt{f} : +, \texttt{g} : \boxtimes, \texttt{h} : +, \_ : \boxtimes\}$ will be split into $\{\texttt{f} : +, \texttt{g} : \boxtimes\}$ and $\{\texttt{h} : +, \_ : \boxtimes\}$; the function will then be applied, and Rule GLUE-FRAME-$\circledast$ will reinsert the missing part, resulting in the object $\{\texttt{f} : +, \texttt{g} : +, \texttt{h} : +, \_ : \boxtimes\}$. To get a similar precise result, TAJS needs to specify that some fields are unchanged by the function using the *modified* component of abstract store values. We believe our approach to be simpler in this aspect.

Both TAJS and the formalisation of Cox et al. use summary nodes. For instance, TAJS provides two abstract locations for each allocation site: a singleton location l and a summary node k. The singleton location tracks the last object allocated from this site, and the summary node tracks all the other. Our approach enables this choice, but is more generic: we enable the use of heuristics which are not based on the allocation site. We also enable to track several singleton locations, and to summarize locations using any heuristic. In particular, the heuristic of TAJS can be expressed in our framework. We thus believe our approach to be very general and to be able in the long run to relate already existing analysers to JSCert or other pretty-big-step semantics.

In this chapter, we have presented a formalisation of separation logic based on the pretty-big-step formalism, which has been introduced in the previous two chapters. Separation logic does not interact well with abstract interpretation, but we have identified an interesting problem in the interactions between the frame rule and the weakening rule of abstract interpretation. We claim that membranes can solve this problem by precisely storing the rewritings performed by the frame rules. There is still room for improvement, notably in the proof of soundness of our approach. To the extend of our knowledge, the notion of membrane is a novel approach. We have seen that they are expressive enough to express shape properties and we think that they are worth further investigation.

We think that the formalism of Chapters 4 and 5 enables the definition and the proof of abstract semantics in a guided and principled way. We have shown that it enables to express various kinds of abstract analyses, including abstract domains from separation logic. We also think that this approach helps to formally relate analysers to certified abstract semantics, and that it is a significant step towards the certification of real-world analysers for JavaScript.

Conclusion

In this dissertation, we have seen how complex JavaScript is. The complexity of JavaScript does not come from its memory model, but from all the exceptions introduced by the JavaScript language itself. This is interesting as it is a completely different kind of complexity than this of languages like C♯minor, notably analysed by the Verasco certified analyser [Jou+15]. The complexity of C♯minor resides in its memory model, in particular, its pointer arithmetic. The memory model of JavaScript is comparably simple (see Section 1.2.3). What makes the difficulty of JavaScript is the size of its semantics. For comparison, the Cminor semantics [Ler06] contains about the same number of derivation rules than our O'While language (see Figures 4.15, 4.17, and 6.3). JavaScript is much bigger: JSCert contains almost a thousand semantic rules and does not yet cover the full JavaScript semantics (see Section 2.4.2).

Such sizes raise several issues. One of these issues is that it is difficult to trust any program analyser for JavaScript, as it is highly probable that one of these corner cases has been missed by programmers. We built JSCert with the explicit goal to serve as a basis for the Coq certification of analyses on the JavaScript language. We invested a large amount of effort to make JSCert as trustable as it could be. The JSCert project relies on two main trust sources: the JSCert specification has been written to closely correspond to the official specification, and JSCert comes with the JSRef interpreter, which has been run against JavaScript test suites. These two certification techniques make JSCert a highly trustable semantics. In particular, JSCert can now be used as a new trust source for analyses and other tools. The JSCert project is still an ongoing project—as Section 2.7.5 shows—and we expect it to evolve with the updates of JavaScript.

JSCert is a large semantics. In particular, the amount of work to apply a traditional approach for building abstract interpreters appears to be overwhelming. We have thus introduced a new way of building abstract analysers. We completed the approach of Schmidt (see Section 3.3) to make it fit the framework of JSCert, namely the pretty-big-step format. Our approach aims at reducing the cost of defining an abstract semantics, as well as proving it sound. To this end, we built a Coq framework to define certified abstract semantics. In our framework, each rules is *independently* abstracted. In particular, abstract rules do not have to consider how different concrete rules can interact with each other. Our framework is then able to build an abstract semantics from these rules, as explained in Section 4.4.2. The soundness proof is similar: each pair of abstract and concrete rules are locally proven sound, and our framework ensures that the whole abstract semantics is sound (see Theorem 5.1 of Section 5.3). Note that we only focus on abstract *semantics* and not abstract *interpreters*. Indeed, given the size of the JSCert semantics, we consider that proving an abstract semantics sound is already a huge work. We consider that building abstract interpreters is a second step in our project. Such interpreters have to use heuristics to efficiently find invariants. Building efficient real-world interpreters is a very different task than building abstract semantics, both being especially difficult in the case of JavaScript given the size of its semantics.

We have instantiated our framework with a memory domain to analyse our O'While language. This memory model is based on separation logic, which we believe is a powerful framework to ensure the modularity of an analysis. Separation logic is based on a special rule, called the frame rule (see Section 6.2). This rule is known to have some issues when interacting with abstract interpretation, in particular in the way in which separation logic treats identifiers in formulae. We proposed a novel approach to protect these identifiers, and more generally to enable global reasoning in our framework based on local actions. This approach is based on the notion of membranes. Membranes have been designed to propagate the local renamings performed on formulae. This work led us to precisely identify the interaction between the frame rule of separation logic and the weakening rule of abstract interpretation, as well as proposing a solution to mix these two rules.

This thesis has resulted in several Coq developments [Bod16]. In particular the JSCert specification and the JSRef interpreter, as well as its proof of correctness. We also provide a precise formalisation of the pretty-big-step format. This led us to formalise the notions of concrete and abstract semantics in their general case. This formalisation has been instantiated into several languages, and has been used to certify abstract analysers. These analysers have been extracted and can thus be run (for instance online [Bod16]). These abstract interpreters were able to produce non-trivial results. We have also formalised a large part of the domains of separation logic presented in Chapter 6.

Perspectives

This thesis opens various paths for further works. Of course, the unfinished development of Chapter 6 has to be continued. In particular, we would like to explore more flexible interactions between the weakening rule and the frame rule (see Section 6.4.5).

This would offer opportunities to further mix abstract interpretation with separation logic. In particular, we believe that membranes can be extended to hold more kinds of rewritings, or could be used in combination with other structures of separation logic. An interesting direction would be to build a vertical frame rule, which rewrites entire abstract derivations. As mentioned in Section 6.4.4, format 1 and 2 rules have not been minimally specified in our O'While language. This is due to the way which the frame rule applies: from an already built semantic triple $\Phi, t \Downarrow^\sharp \Phi'$, we can infer $\phi_c \circledast \Phi, t \Downarrow^\sharp \phi_c \circledast \Phi'$. This rewriting is made after the initial semantic triple has been defined. The idea of a vertical frame rule would be to enable the rewriting of abstract derivations during their construction. For instance, we would like to be able to derive the second derivation below from the first. We could then minimal specify the format 1 Rule red-while-2(e, s)—for instance only on the membraned formula $(- \mid \mathit{emp}, \eta_e, \eta_c)$—and apply this vertical frame rule to complete the needed resources for the above derivation. The current formalism does not enable this, and it would be an interesting extension.

$$\dfrac{\dfrac{\vdots}{(- \mid \mathit{emp}, \eta_e, \eta_c)\,, \mathit{while}\ (e > 0)\ s \Downarrow \Phi}}{(- \mid \mathit{emp}, \eta_e, \eta_c)\,, \mathit{while}_2\ (e > 0)\ s \Downarrow \Phi}\ \textsc{red-while-2}(e, s)$$

$$\dfrac{\dfrac{\vdots}{\phi \circledast (- \mid \mathit{emp}, \eta_e, \eta_c)\,, \mathit{while}\ (e > 0)\ s \Downarrow \Phi}}{\phi \circledast (- \mid \mathit{emp}, \eta_e, \eta_c)\,, \mathit{while}_2\ (e > 0)\ s \Downarrow \Phi}\ \textsc{red-while-2}(e, s)$$

This thesis provides two formalisms which are not yet connected. On one hand, we have indeed formalised a pretty-big-step semantics of JavaScript, namely JSCert. On the other hand, we have formalised how to build and prove sound an abstract semantics from a pretty-big-step semantics. A natural step would thus be to apply the formalism of Chapters 4 and 5 to JSCert. Such a task would be an interesting exercise, but it would be long—even in the systematic, almost mechanical approach developed in this thesis. Indeed, the pretty-big-step style of JSCert is slightly different from our pretty-big-step formalisation. In particular JSCert does not specifies what has to be considered as a syntax element, and what has to be considered as a semantic element. For instance, we show below an extract of JSCert from Program 2.8. This rule is to be compared to Rule red-while-2(e, s) (see Figure 4.2) also shown below. In this last rule, the terms e and s are identified as syntactic elements as they appear in the rule name. Similarly, the elements H, H_e, ℓ_e, and ℓ_c are semantic elements. In the Coq rule `red_while_2e_ii_false`, no separation is made between syntactic elements like `e1` and `t2` and semantic elements like `S` and `C`. The translation from JSCert to the formalism of Chapter 4 is thus not trivial.

```
| red_while_2e_ii_false : forall S C labs e1 t2 rv R o,
     res_type R = restype_normal →
     red_stat S C (stat_while_1 labs e1 t2 rv) o →
     red_stat S C (stat_while_6 labs e1 t2 rv R) o
```

$$\frac{H, H_e, \ell_e, \ell_c, \textit{while}\ (e > 0)\ s \Downarrow r}{H, H_e, \ell_e, \ell_c, \textit{while}_2\ (e > 0)\ s \Downarrow r}\ \textsc{red-while-2}(e, s)$$

Additionally, JSCert does not contain any transfer functions: the JSCert specification is defined by a predicate. We do not foresee any major difficulties in a translation of JSCert to our pretty-big-step formalism. It would however take a large effort, and other possibilities could be envisaged. For instance, we could make use of the JSExplain project (see Section 2.8). This project aims at providing a specification of JSCert and JSRef in a unique format, translatable to a Coq specification and an OCaml program. The JSExplain project is on-going work, and we expect it to eventually fit in the present framework.

From such a version of JSCert, we could then define abstract domains to analyse JavaScript. Such domains can be extensions of the domains defined in Chapter 6, but can also be completely different. The longer part of this step will be to abstract and prove each abstract transfer function for each rule in the chosen domains. We have however made sure to make this step as simple as possible (as it does not require to consider the interactions between concrete rules) and we expect this step to be partially automatable. Once an abstract semantics has been defined, a challenging step would be to certify already existing JavaScript analysers, such as TAJS [JMT09]. We expect the proof of the soundness of such analyser to require less effort by our means than by a direct approach.

Alternatively, our approach could be applied by providing more precise domains for the O'While language, then use the formalism of λ_{JS} or JSExplain to analyse a JavaScript program. Indeed, each of these two projects translates a JavaScript program into a simpler intermediary language. Such a language appears to be close to O'While and we do not expect the formalisation of such a language in our framework to be difficult. This would provide a formally verified analyser for this small language. To analyse a JavaScript program, we would then translate it into this language to analyse the resulting program.

In this dissertation, we have focussed on JavaScript. But the approach of Chapters 4, 5, and 6 could be applied to any semantics expressed in pretty-big-step. It can thus serve as a basis to analyse other kinds of languages. The pretty-big-step format is currently not spread among language semantics. It is has however been shown that big-step semantics can be represented as small-step semantics manipulating continuations [AB07]. We thus expect that our formalism can be adapted to small-step semantics, which would provide a large scale of semantics to be analysed.

Bibliography

[AB07] Andrew W. Appel and Sandrine Blazy. 'Separation Logic for Small-Step Cminor'. In: *Theorem Proving in Higher Order Logics*. 2007 (cit. on p. 182).

[AGD05] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 'Towards Type Inference for JavaScript'. In: *ECOOP*. 2005 (cit. on p. 31).

[Alg+10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 'Fences in Weak Memory Models'. In: *CAV*. 2010 (cit. on p. 30).

[App11] Andrew W. Appel. 'Verified Software Toolchain'. In: *ESOP*. 2011 (cit. on p. 30).

[AR99] Thorsten Altenkirch and Bernhard Reus. 'Monadic Presentations of Lambda Terms Using Generalized Inductive Types'. In: *International Workshop on Computer Science Logic*. 1999 (cit. on p. 155).

[Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 'Mathematizing C++ Concurrency'. In: *POPL*. 2011 (cit. on p. 31).

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. 'Understanding Typescript'. In: *ECOOP*. 2014 (cit. on p. 6).

[BBW15] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 'A Concrete Memory Model for CompCert'. In: *ITP*. 2015 (cit. on p. 30).

[BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. 'SLAyer: Memory Safety for Systems-Level Code'. In: *CAV*. 2011 (cit. on p. 152).

[BCO04] Josh Berdine, Cristiano Calcagno, and Peter W O'hearn. 'A Decidable Fragment of Separation Logic'. In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. 2004 (cit. on p. 152).

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 'Smallfoot: Modular Automatic Assertion Checking with Separation Logic'. In: *International Symposium on Formal Methods for Components and Objects*. 2005 (cit. on pp. 150, 152).

[BDG13] Mark Batty, Mike Dodds, and Alexey Gotsman. 'Library Abstraction for C/C++ Concurrency'. In: *POPL*. 2013 (cit. on p. 31).

[BDM13] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 'Language-Based Defenses Against Untrusted Browser Origins'. In: *USENIX Security Symposium*. 2013 (cit. on p. 32).

[Ber09] Yves Bertot. 'Structural Abstract Interpretation: A Formal Study Using Coq'. In: *Language Engineering and Rigorous Software Development*. 2009 (cit. on p. 124).

[BG14] James Brotherston and Nikos Gorogiannis. ‘Cyclic Abduction of Inductively Defined Safety and Termination Preconditions’. In: *SAS*. 2014 (cit. on pp. 152, 175).

[Bis+06] Steve Bishop, Matthew Fairbairn, Michael Norrish, et al. ‘Engineering with Logic: HOL Specification and Symbolic-Evaluation Testing for TCP Implementations’. In: *POPL*. 2006 (cit. on p. 30).

[BJS15b] Martin Bodin, Thomas Jensen, and Alan Schmitt. ‘Certified Abstract Interpretation with Pretty-Big-Step Semantics’. In: *CPP*. 2015 (cit. on p. 89).

[BL09] Sandrine Blazy and Xavier Leroy. ‘Mechanized Semantics for the Clight Subset of the C Language’. In: *Journal of Automatic Reasoning* (2009) (cit. on p. 30).

[Bod+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, et al. ‘A Trusted Mechanised JavaScript Specification’. In: *POPL*. 2014 (cit. on pp. 25, 55, 59).

[Cac+05] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. ‘Extracting a Data Flow Analyser in Constructive Logic’. In: *Theorical Computer Science* (2005) (cit. on p. 76).

[Cal+09] Cristiano Calcagno, Dino Distefano, Peter O’Hearn, and Hongseok Yang. ‘Compositional Shape Analysis by Means of Bi-Abduction’. In: *POPL*. 2009 (cit. on p. 150).

[CC77a] Patrick Cousot and Radhia Cousot. ‘Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints’. In: *POPL*. 1977 (cit. on pp. xv, 2, 67, 69, 124, 133).

[CC77b] Patrick Cousot and Radhia Cousot. ‘Static Determination of Dynamic Properties of Generalized Type Unions’. In: *ACM SIGPLAN Notices*. 1977 (cit. on p. 72).

[CC92] Patrick Cousot and Radhia Cousot. ‘Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation’. In: *International Symposium on Programming Language Implementation and Logic Programming*. 1992 (cit. on p. 94).

[CCR14] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. ‘Automatic Analysis of Open Objects in Dynamic Language Programs’. In: *SAS*. 2014 (cit. on p. 177).

[CDV09] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. ‘Bi-Abductive Resource Invariant Synthesis’. In: *APLAS*. 2009 (cit. on p. 152).

[CF58] Haskell Brooks Curry and Robert Feys. *Combinatory Logic*. North-Holland, 1958 (cit. on p. 28).

[CH88] Thierry Coquand and Gerard Huet. ‘The Calculus of Constructions’. In: *Information and Computation* (1988) (cit. on pp. 28, 53).

[Cha13] Arthur Charguéraud. ‘Pretty-Big-Step Semantics’. In: *Programming Languages and Systems*. 2013 (cit. on pp. 27, 43).

[CHJ12] Ravi Chugh, David Herman, and Ranjit Jhala. ‘Dependent Types for JavaScript’. In: *OOPSLA*. 2012 (cit. on p. 31).

[Chu+09] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. ‘Staged Information Flow for JavaScript’. In: *PLDI*. 2009 (cit. on p. 31).

[Cio13] Ştefan Ciobâcă. ‘From Small-Step Semantics to Big-Step Semantics, Automatically’. In: *Integrated Formal Methods*. 2013 (cit. on p. 28).

[Con16] The Unicode Consortium, ed. *The Unicode Standard*. 1991–2016 (cit. on p. 8).

[Cor12] Renato Corsetti. 'The Movement for Esperanto: Between Creolization and the Report Grin / La movado por Esperanto: inter kreoliĝo kaj la Raporto Grin'. In: *InKoj. Philosophy & Artificial Languages, New series* (2012) (cit. on p. 8).

[Cou99] Patrick Cousot. 'The Calculational Design of a Generic Abstract Interpreter'. In: *Calculational System Design*. IOS Press, 1999 (cit. on pp. 67, 76).

[Cow97] John Woldemar Cowan. *The Complete LOJBAN Language*. Logical Language Group, 1997 (cit. on p. 8).

[COY07] Cristiano Calcagno, Peter W O'Hearn, and Hongseok Yang. 'Local Action and Abstract Separation Logic'. In: *LICS*. IEEE. 2007 (cit. on p. 152).

[CP10] David Cachera and David Pichardie. 'A Certified Denotational Abstract Interpreter'. In: *ITP*. 2010 (cit. on pp. 81, 124).

[DE97] Sophia Drossopoulou and Susan Eisenbach. 'Is the JAVA Type System Sound?' In: *FOOL4*. 1997 (cit. on p. 30).

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W Appel. 'A Fresh Look at Separation Algebras and Share Accounting'. In: *APLAS*. 2009 (cit. on p. 151).

[DSS13] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 'Meta-Theory à la Carte'. In: *POPL*. 2013 (cit. on p. 127).

[Dur+14] Zakir Durumeric, James Kasten, David Adrian, et al. 'The Matter of Heartbleed'. In: *Conference on Internet Measurement Conference*. 2014 (cit. on pp. xiii, 1).

[ECM11] ECMA International, ed. *ECMASCRIPT Language Specification. Standard ECMA-262, Edition 5.1*. 2011 (cit. on p. 7).

[ECM99] ECMA International, ed. *ECMASCRIPT Language Specification. Standard ECMA-262, 3rd Edition*. 1999 (cit. on p. 7).

[ER12] Chucky Ellison and Grigore Roşu. 'An Executable Formal Semantics of C with Applications'. In: *POPL*. 2012 (cit. on p. 30).

[Far+04] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. 'Formal Analysis of JAVA Programs in JavaFAN'. In: *CAV*. 2004 (cit. on p. 31).

[Flo67] Robert W. Floyd. 'Assigning Meanings to Programs'. In: *Mathematical Aspects of Computer Science* (1967) (cit. on p. 151).

[FM14] Daniele Filaretti and Sergio Maffeis. 'An Executable Formal Semantics of PHP'. In: *ECOOP*. 2014 (cit. on p. 30).

[Fou+13] Cédric Fournet, Nikhil Swamy, Juan Chen, et al. 'Fully Abstract Compilation to JAVASCRIPT'. In: *POPL*. 2013 (cit. on p. 32).

[Gar+15] Philippa Gardner, Gareth Smith, Conrad Watt, and Thomas Wood. 'A Trusted Mechanised Specification of JAVASCRIPT: One Year On'. In: *CAV*. 2015 (cit. on pp. 59, 63).

[GL09] Salvatore Guarnieri and Benjamin Livshits. 'GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JAVASCRIPT Code'. In: *USENIX Security Symposium*. 2009 (cit. on p. 31).

[GMS12] Philippa Gardner, Sergio Maffeis, and Gareth Smith. 'Towards a Program Logic for JAVASCRIPT'. In: *POPL*. 2012 (cit. on pp. 2, 32, 147, 152, 167).

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 'The Essence of JavaScript'. In: *ECOOP*. 2010 (cit. on p. 32).

[Gua+11] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, et al. 'Saving the World Wide Web from Vulnerable JavaScript'. In: *ISSTA*. 2011 (cit. on p. 31).

[Gur94] Yuri Gurevich. 'Evolving Algebras'. In: *IFIP*. 1994 (cit. on p. 30).

[HF07] David Herman and Cormac Flanagan. 'Status Report: Specifying JavaScript with ML'. In: *ML*. 2007 (cit. on p. 31).

[Hoa69] Charles Antony Richard Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Communications of the ACM* (1969) (cit. on p. 151).

[How80] William A. Howard. 'The Formulae-As-Types Notion of Construction'. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, 1980 (cit. on p. 28).

[HS12] Daniel Hedin and Andrei Sabelfeld. 'Information-Flow Security for a Core of JavaScript'. In: *CSF*. IEEE, 2012 (cit. on p. 31).

[IO01] Samin Ishtiaq and Peter O'Hearn. 'BI as an Assertion Language for Mutable Data Structures'. In: *POPL* (2001) (cit. on pp. xvi, 2, 147, 150).

[Jag+98] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. 'Single and Loving It: Must-Alias Analysis for Higher-Order Languages'. In: *POPL*. 1998 (cit. on p. 176).

[JC09] Dongseok Jang and Kwang-Moo Choe. 'Points-To Analysis for JavaScript'. In: *SAC* (2009) (cit. on pp. 31, 177).

[JL14] Éric Jaeger and Olivier Levillain. 'Mind Your Language(s): A Discussion about Languages and Security'. In: *SPW*. IEEE. 2014 (cit. on p. 7).

[JMT09] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 'Type Analysis for JavaScript'. In: *SAS*. 2009 (cit. on pp. 31, 85, 177, 182).

[Jou+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 'A Formally-Verified C Static Analyzer'. In: *POPL*. 2015 (cit. on pp. xv, 30, 76, 93, 179).

[Jou16] Jacques-Henri Jourdan. 'Verasco: a Formally Verified C Static Analyzer'. PhD thesis. Université Paris Diderot-Paris VII, 2016 (cit. on p. 135).

[Kah87] Gilles Kahn. *Natural semantics*. Springer, 1987 (cit. on p. 27).

[Lal14] Joseph Lallemand. *Towards a Well-Formedness Proof for JavaScript*. report. ENS Cachan, 2014 (cit. on pp. 48, 60, 88).

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. 'Towards a Mechanized Metatheory of Standard ML'. In: *POPL*. 2007 (cit. on p. 30).

[Ler06] Xavier Leroy. 'Formal Certification of a Compiler Back-End or: Programming a Compiler With a Proof Assistant'. In: *POPL*. 2006 (cit. on p. 179).

[Mat16] Marek Materzok. 'Certified Desugaring of Javascript Programs using Coq'. In: CoqPL (2016) (cit. on pp. 32, 66).

[Max88] Dan Maxwell. 'On the Acquisition of Esperanto'. In: *Studies in Second Language Acquisition* (1988) (cit. on p. 8).

[McB02] Conor McBride. 'Elimination with a Motive'. In: *Types for Proofs and Programs.* 2002 (cit. on p. 121).

[Mil+97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of Standard ML: revised 1997.* The MIT Press, 1997 (cit. on p. 31).

[Min06a] Antoine Miné. 'Symbolic Methods to Enhance the Precision of Numerical Abstract Domains'. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation.* 2006 (cit. on p. 93).

[Min06b] Antoine Miné. 'The Octagon Abstract Domain'. In: *Higher-Order and symbolic computation* (2006) (cit. on pp. xv, 72).

[MJ08] Jan Midtgaard and Thomas Jensen. 'A Calculational Approach to Control-Flow Analysis by Abstract Interpretation'. In: *SAS.* 2008 (cit. on pp. 76, 102).

[MMT08] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 'An Operational Semantics for JavaScript'. In: *APLAS.* 2008 (cit. on p. 31).

[MMT09] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 'Isolating JavaScript with Filters, Rewriting, and Wrappers'. In: *ESORICS.* 2009 (cit. on p. 31).

[MMT10] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 'Object Capabilities and Isolation of Untrusted Web Applications'. In: *SP.* IEEE, 2010 (cit. on pp. 1, 7, 31).

[MR05] Laurent Mauborgne and Xavier Rival. 'Trace Partitioning in Abstract Interpretation Based Static Analyzers'. In: *ESOP.* 2005 (cit. on p. 135).

[MS06] Matthew Might and Olin Shivers. 'Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting'. In: *ICFP.* 2006 (cit. on p. 176).

[MT09a] Sergio Maffeis and Ankur Taly. 'Language-Based Isolation of Untrusted JavaScript'. In: *CSF.* IEEE. 2009 (cit. on pp. 1, 7).

[MT09b] Sergio Maffeis and Ankur Taly. 'Language-Based Isolation of Untrusted JavaScript'. In: *CSF.* IEEE, 2009 (cit. on p. 31).

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, 1997 (cit. on pp. 26, 30).

[Nor98] Michael Norrish. 'Formalising C in HOL'. PhD thesis. Computer Laboratory, University of Cambridge, 1998 (cit. on p. 30).

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science. Springer, 2002 (cit. on p. 25).

[ORY01] Peter O'Hearn, John C. Reynolds, and Hongseok Yang. 'Local Reasoning about Programs that Alter Data Structures'. In: *CSL.* 2001 (cit. on pp. xvi, 2, 147, 150).

[Owe08] Scott Owens. 'A Sound Semantics for OCaml Light'. In: *Programming Languages and Systems.* 2008 (cit. on p. 30).

[Par69] David Park. 'Fixpoint Induction and Proofs of Program Properties'. In: *Machine Intelligence.* Edinburgh University Press, 1969 (cit. on p. 121).

[PBC06] Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. 'Variables as Resource in Hoare Logics'. In: *LICS.* IEEE. 2006 (cit. on p. 153).

[Pic05] David Pichardie. 'Interprétation abstraite en logique intuitionniste: extraction d'analyseurs JAVA certifiés'. PhD thesis. Université Rennes 1, 2005 (cit. on p. 67).

[Pic08] David Pichardie. 'Building Certified Static Analysers by Modular Construction of Well-Founded Lattices'. In: *FICS*. ENTCS. 2008 (cit. on p. 81).

[PK87] Gordon Plotkin and Gilles Kahn. 'A Structural Approach to Operational Semantics'. In: *STACS*. 1987 (cit. on p. 27).

[PLR11] Changhee Park, Hongki Lee, and Sukyoung Ryu. 'An Empirical Study on the Rewritability of the with Statement in JAVASCRIPT'. In: *FOOL*. 2011 (cit. on p. 31).

[PLR12] Changhee Park, Hongki Lee, and Sukyoung Ryu. 'SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMASCRIPT'. In: *FOOL*. 2012 (cit. on p. 31).

[PM14] Casper Bach Poulsen and Peter D. Mosses. 'Deriving Pretty-Big-Step Semantics from Small-Step Semantics'. In: *Programming Languages and Systems*. 2014 (cit. on p. 28).

[Pol+11a] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 'ADsafety: Type-Based Verification of JAVASCRIPT Sandboxing'. In: *USENIX Conference on Security*. 2011 (cit. on pp. 1, 7, 32).

[Pol+11b] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 'Type-Based Verification of JAVASCRIPT Sandboxing'. In: *USENIX Conference on Security*. 2011 (cit. on p. 25).

[Pol+12a] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 'A Tested Semantics for Getters, Setters, and `eval` in JAVASCRIPT'. In: *DLS* (2012) (cit. on p. 32).

[PS99] Frank Pfenning and Carsten Schürmann. 'System Description: Twelf - A Meta-Logical Framework for Deductive Systems'. In: *CADE*. 1999 (cit. on p. 30).

[PSC09] Phu H. Phung, David Sands, and Andrey Chudnov. 'Lightweight Self Protecting JAVASCRIPT'. In: *ASIACCS*. 2009 (cit. on p. 31).

[PSR15] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 'KJS: A Complete Formal Semantics of JAVASCRIPT'. In: *PLDI*. 2015 (cit. on p. 32).

[Rey02] John C. Reynolds. 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: *LICS*. 2002 (cit. on pp. xvi, 2, 147, 150).

[Rey08] John C. Reynolds. 'An Introduction to Separation Logic'. In: *Engineering Methods and Tools for Software Safety and Security* (2008) (cit. on pp. xvi, 2, 147, 150).

[Ric+10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 'An Analysis of the Dynamic Behavior of JAVASCRIPT Programs'. In: *PLDI*. 2010 (cit. on p. 19).

[Ric+11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 'The `eval` that Men Do. A Large-Scale Study of the Use of `eval` in JAVASCRIPT Applications'. In: *ECOOP*. 2011 (cit. on pp. 19, 31).

[Ric53] Henry Gordon Rice. 'Classes of Recursively Enumerable Sets and Their Decision Problems'. In: *Transactions of the American Mathematical Society* (1953), pp. 358–366 (cit. on p. 67).

[RL09] Grigore Roşu and Dorel Lucanu. ‘Circular Coinduction: A Proof Theoretical Foundation’. In: *International Conference on Algebra and Coalgebra in Computer Science*. 2009 (cit. on p. 29).

[RM07] Xavier Rival and Laurent Mauborgne. ‘The Trace Partitioning Abstract Domain’. In: *TOPLAS* (2007) (cit. on p. 135).

[RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. ‘An Overview of the $\mathbb{K}$ Semantic Framework’. In: *Journal of Logic and Algebraic Programming* (2010) (cit. on p. 30).

[Sch95] David A. Schmidt. ‘Natural-Semantics-Based Abstract Interpretation (preliminary version)’. In: *SAS*. 1995 (cit. on pp. 77, 102).

[Šev+11] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. ‘Relaxed-Memory Concurrency and Verified Compilation’. In: *POPL*. 2011 (cit. on p. 30).

[Sew+10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, et al. ‘Ott: Effective Tool Support for the Working Semanticist’. In: *Journal of Functional Programming* (2010) (cit. on p. 30).

[SO08] Matthieu Sozeau and Nicolas Oury. ‘First-Class Type Classes’. In: *Theorem Proving in Higher Order Logics* (2008) (cit. on p. 79).

[SRW98] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. ‘Solving Shape-Analysis Problems in Languages with Destructive Updating’. In: *TOPLAS* (1998) (cit. on pp. 147, 170).

[Ste+85] IEEE, ed. *An American National Standard-IEEE Standard for Binary Floating-Point Arithmetic*. Std 754. 1985 (cit. on p. 8).

[Ste81] David Stevenson. ‘A Proposed Standard for Binary Floating-Point Arithmetic’. In: *Computer* (1981) (cit. on p. 8).

[Sym99] Don Syme. ‘Proving Java Type Soundness’. In: *Formal Syntax and Semantics of Java*. 1999 (cit. on p. 30).

[Tal+11] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. ‘Automated Analysis of Security-Critical JavaScript APIs’. In: *SP*. 2011 (cit. on p. 32).

[Tar55] Alfred Tarski. ‘A Lattice-Theoretical Fixpoint Theorem and its Applications’. In: *Pacific journal of Mathematics* (1955) (cit. on p. 122).

[Ter95] Delphine Terrasse. ‘Encoding Natural Semantics in Coq’. In: *International Conference on Algebraic Methodology and Software Technology*. 1995 (cit. on p. 28).

[Thi05] Peter Thiemann. ‘Towards a Type System for Analyzing JavaScript Programs’. In: *ESOP*. 2005 (cit. on p. 31).

[VB14] Jérôme Vouillon and Vincent Balat. ‘From Bytecode to JavaScript: the Js_of_ocaml Compiler’. In: *Software: Practice and Experience* (2014) (cit. on p. 5).

[VM11a] David Van Horn and Matthew Might. ‘Abstracting Abstract Machines: A Systematic Approach to Higher-Order Program Analysis’. In: *Communications of the ACM* (2011) (cit. on p. 102).

[VM11b] David Van Horn and Matthew Might. ‘An Analytic Framework for JavaScript’. In: *CoRR* (2011) (cit. on p. 150).

[Wad92] Philip Wadler. ‘The Essence of Functional Programming’. In: *POPL*. 1992 (cit. on p. 50).

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. 'Guarded Recursive Datatype Constructors'. In: *POPL*. 2003 (cit. on p. 28).

[Zak11] Alon Zakai. 'Emscripten: An LLVM-To-JavaScript Compiler'. In: *OOPSLA*. 2011 (cit. on p. 5).

External Links

[05] Mozilla, ed. *Mozilla Firefox's Standard Built-in Objects*. 2005. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects#Non-standard_objects` (cit. on p. 23).

[BJS15a] Martin Bodin, Thomas Jensen, and Alan Schmitt. *Certified Abstract Interpretation with Pretty Big-Step Semantics: Coq Development Files and Analyzers Results*. 2015. URL: `http://ajacs.inria.fr/coq/cpp2015/` (cit. on pp. 89, 91).

[Bod+12] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, et al. *JSCert: Certified JavaScript*. 2012. URL: `http://jscert.org/` (cit. on p. 25).

[Bod16] Martin Bodin. *Thesis Companion*. 2016. URL: `http://people.irisa.fr/Martin.Bodin/doktorigxo/companion.html?lang=en` (cit. on pp. 2, 4, 38, 78, 95, 122, 125, 147, 149, 168, 169, 180).

[Cha10] Arthur Charguéraud. *TLC: A Non-Constructive Library for Coq Based on Typeclasses*. 2010. URL: `http://www.chargueraud.org/softs/tlc/` (cit. on pp. 53, 79, 120).

[CHP+84] Thierry Coquand, Gérard Huet, Christine Paulin, et al. *the Coq Proof Assistant*. 1984. URL: `https://coq.inria.fr/` (cit. on pp. xiv, 2, 25).

[Cle12] Xavier Clerc. *Bisect*. 2012. URL: `http://bisect.x9c.fr/` (cit. on p. 61).

[Cro08] Douglas Crockford. *ADsafe: Making JavaScript safe for advertising*. 2008. URL: `http://www.adsafe.org` (cit. on pp. 6, 25, 89).

[CSW16] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. *Interactive Debugger for the JavaScript Specification*. 2016. URL: `https://github.com/jscert/jsexplain` (cit. on p. 62).

[ECM10] ECMA International. *Test262*. 2010. URL: `https://github.com/tc39/test262` (cit. on pp. 26, 32, 59).

[ECM15] ECMA International. *Contributing to ECMAScript*. 2015. URL: `https://github.com/tc39/ecma262/blob/master/CONTRIBUTING.md#contributing-to-ecmascript` (cit. on p. 8).

[Fre10] Free Software Foundation. *C Language Testsuites: "C-Torture" version 4.4.2*. 2010. URL: `http://gcc.gnu.org/onlinedocs/gccint/C-%20Tests.html` (cit. on p. 30).

[Fug11] Dave Fugate. *Test262 Bug 56*. 2011. URL: `https://bugs.ecmascript.org/show_bug.cgi?id=56` (cit. on p. 60).

[Goo02] Google, ed. *Google API Explorer*. 2002. URL: `https://developers.google.com/apis-explorer` (cit. on p. 6).

[Gro11] The WG14 Working Group. *ISO/IEC 9899:2011*. 2011. URL: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853` (cit. on pp. 25, 31).

[Hid12] Ariya Hidayat. *ECMAScript Parsing Infrastructure for Multipurpose Analysis.* 2012. URL: http://esprima.org/ (cit. on pp. 38, 52).

[HWZ13] David Herman, Luke Wagner, and Alon Zaka. *asm.js.* Ed. by Mozilla. 2013. URL: http://asmjs.org (cit. on p. 6).

[Kle13] Martin Kleppe. *JSFuck.* 2013. URL: http://jsfuck.com (cit. on p. 14).

[Ler+08] Xavier Leroy, Sandrine Blazy, Zaynah Dargaye, and Jean-Baptiste Tristan. *CompCert: Compilers you can formally trust.* 2008. URL: http://compcert.inria.fr/ (cit. on pp. 25, 30).

[LLT10] Brian Leroux, Andrew Lunny, and David Trejo, eds. *wtfjs.* 2010. URL: http://wtfjs.com (cit. on pp. 7, 21).

[Mad15] David Madore. *Why English sucks as the language for international and scientific communication.* 2015. URL: http://www.madore.org/~david/weblog/d.2015-03-20.2284.html (cit. on pp. 2, 7).

[Mel12] Guillaume Melquiond. *Floats for Coq 2.1.0.* 2012. URL: http://flocq.gforge.inria.fr/ (cit. on pp. 52, 53).

[Moz13] Mozilla. *Mozilla Automated JavaScript Tests.* 2013. URL: https://developer.mozilla.org/en-US/docs/SpiderMonkey/Running_Automated_JavaScript_Tests (cit. on pp. 26, 32).

[PGG] Erin Piateski, Jevgenij Gaus, and Neringa Gaus. *Lernu!* URL: http://lernu.net (cit. on p. 8).

[Pol+12b] Joe Gibbs Politz, Benjamin S. Lerner, Hannah Quay-de la Vallee, et al. *Mechanized λ_{JS}.* 2012. URL: http://blog.brownplt.org/2012/06/04/lambdajs-coq.html (cit. on p. 32).

[Sch12] Till Schneidereit. *Convert some array extras to self-hosted js implementations.* 2012. URL: https://hg.mozilla.org/mozilla-central/rev/5593cd83590e (cit. on p. 62).

[Thea] The es-discuss community. *Email Thread: "for-in", shadowing and deleting properties.* URL: https://mail.mozilla.org/pipermail/es-discuss/2013-May/030625.html (cit. on p. 63).

[Theb] The es-discuss community. *Email Thread: loop unrolling and completion values in ES6.* URL: https://mail.mozilla.org/pipermail/es-discuss/2015-April/042351.html (cit. on pp. 8, 34).

[The13] The JSCert Team. *Bisect report of JSRef when running the test262 test suite.* 2013. URL: http://jscert.org/popl14/Bisect_report.html (cit. on p. 61).

[Var12a] Various developers. *Reported bugs in Mozilla.* 2010–2012. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=819125 (cit. on p. 60).

[Var12b] Various developers. *Reported bugs in V8.* 2010–2012. URL: http://code.google.com/p/v8/issues/detail?id%20=%20%7B705,2446%7D (cit. on p. 60).

[Var12c] Various developers. *Reported bugs in WebKit.* 2010–2012. URL: https://bugs.webkit.org/show_bug.cgi?id%20=%20%7B38970,104309%7D (cit. on pp. 60, 64).

Other Citations

[Asi85] Isaac Asimov. *Robots and Empire*. Doubleday New York, 1985 (cit. on p. 147).

[Aud97] Michèle Audin. *Conseils aux auteurs de textes mathématiques*. 1997. URL: `http://irmasrv1.u-strasbg.fr/~maudin/publications.html` (cit. on p. 1).

[FD70] André Franquin and Jean De Mesmaeker. *Gala de gaffes à gogo*. Dupuis, 1970, p. 20 (cit. on p. 133).

[Kal13] Nanne Kalma. *Tohuvabohuo*. Ed. by Vinilkosmo. 2013 (cit. on p. 3).

[Rou09] Gilles Roussel. 'Le Repos du guerrier'. In: *Quelques minutes avant la fin du monde*. Delcourt, 2009 (cit. on p. 25).

[Wei10] Zachary Weinersmith. *Saturday Morning Breakfast Cereal*. 2010. URL: `http://www.smbc-comics.com/comic/2010-02-23` (cit. on p. 67).

[Yud15] Eliezer Yudkowsky. *Harry Potter and the Methods of Rationality*. 2015. URL: `http://hpmor.com/` (cit. on p. 89).