THÈSE

présentée à

l'ÉCOLE NORMALE SUPÉRIEURE

pour l'obtention du titre de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE EN INFORMATIQUE

Arlen Cox

13 avril 2015

Parametric Heap Abstraction for Dynamic Language Libraries

Domaines Abstraits Paramètriques pour la Représentation du Tas et l'Analyse Statique des Langages Dynamiques.

Président:	Ahmed Bouajjani *Professeur, University Paris Diderot, France*
Rapporteurs:	Anders Møller *Associate Professor, Aarhus University, Danemark* Jeremy Siek *Associate Professor, Indiana University, États-Unis*
Jury:	Marc Pouzet *Professeur, University Pierre et Marie Curie, France*
Directeurs de thèse:	Xavier Rival *Chargé de Recherche, INRIA Rocquencourt, France* Bor-Yuh Evan Chang *Assistant Professor, University of Colorado Boulder, États-Unis*

École Normale Supérieure
Département d'Informatique

Abstract For commercial development, dynamic languages are growing in popularity. Consequently, dynamic language developers must consider the correctness of their code. Deployment of correct or sufficiently correct code is critical to the success and adoption of that code. However, it is challenging to ensure the correctness of dynamic language code when it is a library. Inputs to dynamic language libraries are often not simple values. They can also be open objects, which permit adding, removing, and iterating over attribute names, and they can be functions that may be called. Furthermore, the result of running library functions on objects is often new objects derived from the input objects.

To ensure the correctness of dynamic language libraries, this dissertation uses static analysis. Static analysis is typically used to infer facts about programs' values, but in these dynamic language libraries, values can be objects and functions. These objects may be unknown and thus have an unknown set of attribute names because they are inputs or these objects may be iteratively derived from other objects. Functions may be stored, called, or wrapped in other functions, regardless of if they are known or not. A static analysis for dynamic language libraries must handle these cases.

To support static analysis of libraries, this dissertation introduces local heap abstractions suitable for representing parts of memory that library code may affect. These abstractions build upon abstractions for sets that enable representing relations between attribute names of otherwise unknown objects. Furthermore, the local reasoning is utilized to abstract the effect of calling unknown functions.

The precision of these abstractions are demonstrated on a range of small, but complex JavaScript-inspired examples. The examples are extracted from libraries such as Traits.js and Google Closure.

Résumé Dans le développement de logiciels, les langages dynamiques sont de plus en plus utilisés. Les développeurs doivent s'assurer de la correction des leurs codes. Le déploiement de code correct ou d'un niveau de qualité suffisant est essentiel au succès du code. Cependant, il est difficile de garantir le correction du code écrit dans un langage dynamique lorsqu'il s'agit d'une bibliothèque. Les paramètres des fonctions des bibliothèques des langages dynamiques sont souvent des valeurs complexes. Ils peuvent être des objets ouverts, qui permettent l'ajout et la suppression de noms, ainsi que l'itération sur des noms d'attributs, et ils peuvent être des fonctions qui peuvent être appliquées. En outre, le résultat de l'application d'une fonction de la bibliothèque à des objets est souvent un nouvel objet provenant des arguments.

Pour s'assurer de la correction des bibliothèques pour les langages dynamiques, cette thèse s'appuie sur l'analyse statique. L'analyse statique est généralement utilisée pour déduire des propriétés des valeurs produites par les programmes. Toutefois, dans ces bibliothèques des langages dynamiques, les valeurs peuvent être des objets et des fonctions. Lors de l'analyse, ces objets peuvent être inconnus et ont un ensemble inconnu de noms d'attributs parce qu'ils sont des paramètres ou parce que ces objets peuvent être itérativement dérivés d'autre objets. Les fonctions peuvent être stockées, appliquées, ou incluses dans d'autres fonctions, indépendamment de s'elles sont connues ou non. Une analyse statique pour les bibliothèques des langages dynamiques doit fonctionner dans de tels cas.

Pour améliorer l'analyse statique des bibliothèques, cette thèse étudie des abstractions du tas local, appropriées pour la représentation des parties du tas que le code de la bibliothèque peut modifier. Ces abstractions sont construites sur des abstractions pour les ensembles qui permettent d'exprimer les relations entre les noms d'attributs des objets inconnus par ailleurs. En outre, les abstractions du tas local permettent de représenter l'effet de l'application de fonctions inconnues.

La précision de l'abstraction est démontrée sur une gamme d'exemples JavaScript qui sont petits mais complexes. Les exemples sont extrait de bibliothèques telles que Traits.js et Google Closure.

Acknowledgements Thank you from the bottom of my heart to everyone who helped me prepare for and write my dissertation. It has been an incredibly challenging and rewarding experience and I would not have made it through without the immeasurable support of my advisors, friends, and family. First, thank you to my advisors Evan, Xavier, and Sriram. You taught me the ropes, got me excited about static analysis, and generally motivated my work from beginning to end. I owe my style to their extensive tutelage, proofreading, and constructive criticism. Thank you Aaron for getting me interested in formal methods. You spotted my interest in formal techniques before I even knew what they were. You taught me logic and your view of the world as logic carries on in my work today. Also, thanks to Josh, Samin, and Christoph for the internship at Microsoft and for getting me out of the biggest slump of my graduate studies. You had hope and faith in me when I did not and continue to support and encourage me to this day.

Thank you to my friends from both Colorado and Paris: Aleks, Caterina, Dan, Devin, Geoff, Huisong, Jiangchao, Joe, Jonathan, Pippijn, Sam, and Yi-Fan. You were there to listen to my ideas, challenge me to do my best work, and to help me out of binds.

Thank you to my family for supporting me through this. To my parents, you were there to talk to me whenever I needed to feel better. You encouraged me through the whole process and guided me to make the right decisions. To Jenny, your never-ending love and support allowed me to make it through some of toughest challenges of my life. You loved me every day no matter where I was in the world and that by itself was enough get me through each challenging day. You were willing to dive in and help when I did not know what to do. I love you so much for this.

Contents

1. Introduction **1**
1.1. Bug Elimination in Dynamic Language Libraries without Clients 2
1.2. The Challenge and Existing Solutions 3
1.3. Handling Non-deterministic Inputs to Libraries 5
1.4. Thesis Statement 8
1.5. Summary 9

2. Motivation: JavaScript Library Verification without Client Code **12**
2.1. Class Library Requirements 14
2.2. Implementing the JavaScript Class Library 14
2.3. Analyzing the JavaScript Class Library 16

3. Overview: Analysis of JavaScript Libraries **18**
3.1. Open-Object-Focused JavaScript Language 18
3.2. Real-World Open-Object-Focused JavaScript 24
3.3. Abstraction: Representation of Program Facts 26
3.4. Example Analysis: JavaScript Class Implementation 27
3.4.1. Unknown Attribute Access 27
3.4.2. Object Attribute Iteration 29
3.4.3. Set Abstraction 32
3.4.4. Unknown Function Calls 32
3.4.5. Bringing the Abstractions Together 34

4. Heap Abstraction: Separation Logic with Open Objects **35**
4.1. Abstract Interpretation Background 35
4.2. Representation: Heap with Open Objects 39
4.2.1. Single-State Heap with Open Objects 39
4.2.2. Two-State Heap with Open Objects 42
4.3. Materialization with Set Abstraction 46
4.4. Reading and Writing in Objects 52
4.5. Automatic Invariant Inference 55
4.6. Related Work 61
4.7. Heap with Open Objects Summary 62

5. Function Abstraction: Desynchronized Separation **64**
5.1. Representation of Desynchronization 65
5.2. Desynchronization with Reachability-based Frame Inference 66

5.3. Introduction Heuristics and Resynchronization 68
5.4. Related Work 70
5.5. Desynchronized Separation Summary 71

6. Discussion of Combined Analysis 72

6.1. Implementing the Combined Analysis 72
6.2. Single-State HOO Performance/Precision 75
6.3. Two-State HOO with Desynchronization 77
6.3.1. Case Study: Class 80
6.3.2. Case Study: Memoization 81
6.4. Boundaries of Analysis and Future Improvements 82

7. Set Abstraction: Relational, First-Class Sets 85

7.1. Overview 87
7.1.1. Set Language 88
7.1.2. Motivating Example 89
7.2. QUIC Graphs 92
7.3. Closure 96
7.3.1. Inference Rules 96
7.3.2. Soundness 99
7.3.3. Complexity 100
7.3.4. Lazy Inference Implementation 100
7.4. Domain Operations 101
7.5. Evaluation 104
7.6. Limitations. 106
7.7. Related Work 107
7.8. Summary of QUIC Graphs 108

8. Conclusions and Future Work 109

A. Full Example Analysis 121

B. Detailed Proofs 127

B.1. HOO Materialization Soundness 127
B.2. HOO Transfer Function Soundness 129
B.3. HOO Join Soundness 131
B.4. HOO Inclusion Soundness 132
B.5. Desynchronization Introduction Soundness 132
B.6. QUIC Graphs Inference Soundness 133
B.7. QUIC Graphs Complexity 134

C. Inclusion Algorithm 135

D. Detailed Tests for Single-State HOO and TAJS 138

E. Detailed Tests for Two-State HOO with Attribute/Value Trackers and Desynchronization 142

F. Detailed Tests for QUIC Graphs 152

1. Introduction

Late bugs are bad and early bugs are good. In fact, studies have shown that the cost of fixing a software bug before deploying it to users is from 5 to 200 times cheaper than fixing a bug after deployment [BP88, BB01]. There are many reasons for this cost disparity, including the cost of redistributing the patched software as well as the cognitive effort to modify an already mostly working system to accommodate a fix. Additionally, the more widely deployed a piece of software is, the more costly it is to make changes: users of the software may depend on subtle behaviors and thus modifications may break compatibility. If compatibility is broken, users may choose not to use the fixes and thus may experience losses due to already fixed bugs, leading to mistrust. In short, software should be developed to optimize the total cost of development, meaning that heavily used software should be carefully developed to ensure that not too many bugs occur after deployment.

Unfortunately, it's not just whole programs that have bugs. Libraries have bugs, too. When bugs occur in libraries, the cost of fixing them after deployment goes up because there are multiple layers of fixes required. For example, the Heartbleed OpenSSL bug[1], which allowed unauthorized access to sensitive information transmitted over an encrypted connection, demonstrated these high costs with respect to a library. Even after the bug was fixed in OpenSSL, all of the client software had to update their version of OpenSSL and then redeploy to their clients. It is not always cost effective to do this and thus even months after the OpenSSL patch was released, 97 percent of major servers remain vulnerable [Ven14]. Additionally, there is now significant mistrust of OpenSSL, leading to two derivative versions LibreSSL and BoringSSL, along with increased popularity of alternatives PolarSSL and MatrixSSL.

What is a well-intentioned library developer (henceforth developer) to do? She may wish to create robust libraries that will not suffer from the problems of OpenSSL, but this is not easy to achieve. The developer must simultaneously balance not having too many bugs with actually shipping a library. To do so, a good approach used by others is to turn to automated techniques. For example, researchers have created a special version of PolarSSL that has been verified to be free of a number of common vulnerabilities (such as buffer overflows) using custom built static analysis[2]. Unfortunately, even well intentioned developers may run into difficulties if automated techniques are unavailable or underdeveloped for their languages of choice as is the case with the popular [Pau07] dynamic languages.

A *dynamic language* is a programming language that has features for and encourages the use of run-time modification of program structures. For example, dynamic languages

[1] http://heartbleed.com/
[2] http://trust-in-soft.com/polarssl-verification-kit/

have *open objects* — objects where attributes (field, property, etc.) names are first class and can be mutated, added to objects, removed from objects, and iterated over. Also dynamic languages do not check types before running the program and often do not require fixed types for any variable at any point in the program. Additionally, dynamic languages typically combine features from both object-oriented and functional programming languages. They are imperative and encourage mutation of data structures and variables rather than reallocation with sharing and binding. Simultaneously, however, they offer first-class functions with closures so that enclosing (mutable) environments are captured along with functions for later use. Examples of dynamic languages include JavaScript, Python, Lua, and Ruby. This dissertation focuses on a dynamic-language-inspired core calculus that offers many of the dynamic-language-specific behaviors from all of the dynamic languages.

1.1. Bug Elimination in Dynamic Language Libraries without Clients

Regardless of whether the language is dynamic or not, the goal of a library developer is to produce a robust, reusable library. A library that is riddled with bugs will likely not be widely adopted and will certainly not be robust. Consequently, the challenge is to identify bugs prior to deploying the library to boost confidence in the library and thus allow it to be heavily reused.

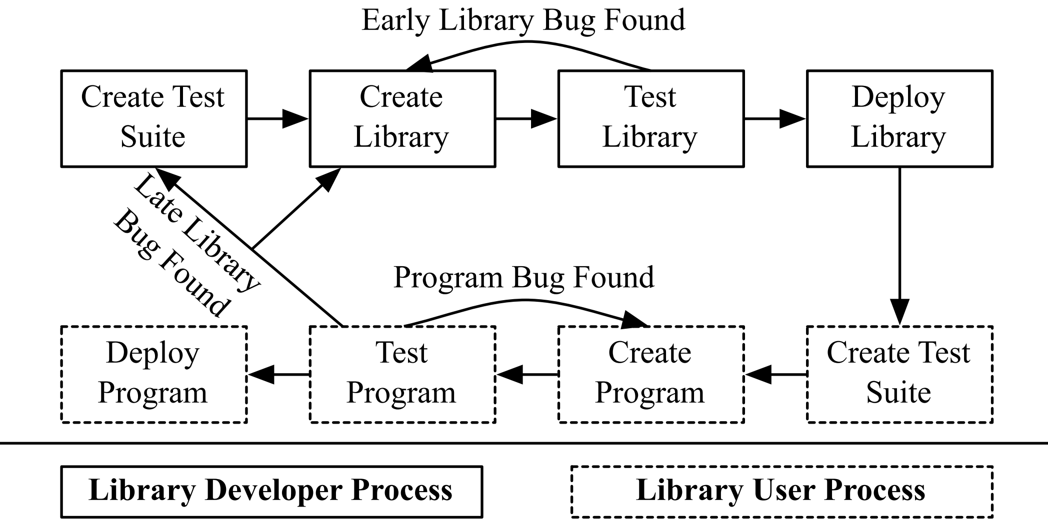


Figure 1.1. – If a library is deployed before it is free of bugs, late bugs may require fixes in and redeployment of the library

Figure 1.1 shows the software development process for both libraries and programs together. The creation of a library starts with a specification of the behavior of the library (formal or informal). From this a developer can create a thorough test suite. Of course,

the creation of a robust test suite is a significant challenge by itself because it requires understanding not only exactly what the library should do, but selecting key instances of inputs that will exhibit all of the potentially buggy behavior of the library. Once the test suite has been completed (assuming a good test-driven development strategy), the library can be written and then tested. Any bugs that the test suite reveals can be fixed and this process can be iterated until there are no longer any bugs worth fixing. At this point the library can be deployed.

Once the library has been deployed, it can be used by arbitrarily many users, who may follow the same process as the library. There is one difference in this process, however. Bugs in a program can either be due to a flaw in the program itself, or a flaw in a library that the program depends upon. If a bug is found to be due to the library, the process becomes much more complicated. Information on the bug must be fed back to the library developer, who will extend the test suite, fix the library, re-test the library, and re-deploy the library. When this happens, however, each user of the library may also want to update to the latest version of the library and then the cycle repeats.

In this figure, there are two library bug detection phases and bugs can be caught at any phase. The early library bug phase is the preferable phase as it eliminates any undue burden on potentially many users and maintains trust. Of course, finding bugs in this phase relies upon specific knowledge of what the library should do (i.e. a specification) and that can be a lofty goal. Alternatively, finding bugs in the late phase relies only on the program test suite, which the library developer does not have to create, but finding out about the bugs may be difficult. Users have to report them.

1.2. The Challenge and Existing Solutions

The challenge addressed in this dissertation is lowering the bar for robust early bug detection for dynamic languages, making it easier to produce reliable libraries. Traditional testing does not meet this goal simply because it is too dependent on developers' abilities to select good tests (examples of a specification). Alternatively, there are several approaches that can help eliminate the need for developers to design good tests: types, random testing, extended static checking, and static program analysis. This section discusses the current state of each of these techniques with regards to JavaScript libraries.

Type Systems Arguably, the most popular way of eliminating bugs in programs is with type systems [Car04]. Most non-dynamic languages have static-type checking that prohibits compilation if the type of a value is inconsistent with the use of that value. Unfortunately, this is not typically the case with dynamic languages. However, some dynamic languages (e.g. TypeScript [BAT14]) have type systems that can help find many simple but problematic bugs, even if the type systems are unsound. Unfortunately, such type systems are incapable of dealing with many of the dynamic features of dynamic languages. For example, in the TypeScript community there is the definitely typed repository [3], which provides assumed type annotations for libraries that are too dynamic

[3] https://github.com/borisyankov/DefinitelyTyped

for TypeScript's built-in type inference. More generally, type systems are arbitrarily expressive [CH88]. The challenge is, as with any question of verification, finding a trade off between developer effort, verification time, and likelihood of success. When type systems become sufficiently expressive, they become similar enough to either extended static checkers or static program analyses that they might as well be considered as such.

Randomized and Directed Testing As an alternative to simple type systems, more robust testing is possible through variants of randomized testing. For example, there are several fuzz testing tools for JavaScript[4]. These tools randomly generate inputs to functions (including randomly generated functions) and then run the functions on those inputs to see if they generate any errors. Of course, this requires a form of specification: the random input values must be restricted to valid inputs and outputs of the library must be checked to ensure that they are correct with respect to the given input. Naturally, such approaches do not guarantee any coverage of the code and they can be very bad for finding rare cases. For example, when dividing 32-bit integers, the odds of randomly selecting the one integer than causes a divide-by-zero error is low.

As opposed to fully random testing, it is possible to use a more directed form of randomized testing approach such as concolic testing. Jalangi [SKBG13], or Kudzu [SAH+10] use SMT solvers [dMB08, BCD+11, CGSS13] to guide the randomized testing to explore a wide range of paths through the program. These SMT solvers produce inputs specially designed to cover a wide variety of paths through the program. Thus, concerns about that random selection of that one bad value are somewhat alleviated. These have been shown to be quite effective at locating many bugs, but, for libraries, they not only require a specification, but because many programs have an unbounded number of paths through the program, they also never give any guarantees about the absence of bugs. Further exacerbating this problem is that they are limited by the capabilities of the SMT solvers, which are not currently well suited for the heap, objects, functions, and strings used heavily by dynamic languages.

Extended Static Checkers While simple type systems can offer robustness, they lack the expressivity to handle the complex programs that library developers often write. Random testers suffer the opposite problem of having the expressivity, but without robustness. By trading away automation, extended static checkers realize the benefits of both. Extended static checkers, such as DJS [CHJ12] or JuS [GNS13] for JavaScript, require not only specification of inputs and outputs for the library, but also loop invariants for the library. *Loop invariants* state what is true about every iteration of a loop. Given sufficiently precise loop invariants for each loop in the program, a constraint solver may be able to automatically verify the specification.

Unfortunately, while this can be an effective way to verify library functionality, the need for loop invariants is problematic. The developer is no longer solely concerned with what goes into and out of the library, but also with invariants of every loop. The inclusion

[4]jsfuzzer: https://code.google.com/p/jsfuzzer/ JavaScript-fuzz: https://github.com/NodeGuy/JavaScript-fuzz

of loop invariants in the specification, in addition to making the specification significantly more complicated, means that the specific implementation of a function is deeply tied to its specification.

Static Program Analysis If robustness, expressivity, and automation are all important, static program analysis is a good option. Of course, there are trade-offs. It is impossible to construct a fully precise, automatic analysis for Turing complete languages [Kle43]. However, it is possible to construct usefully expressive and automated program analyses that are also robust. For example, Astrée [BCC$^+$03] solves this problem for embedded C programs. For non-dynamic languages, it is even possible to develop powerful analyses for library code [FL10, CDOY11]. For dynamic languages, existing analyzers are whole program analyzers [JMT09, LWJ$^+$12, KSW$^+$13, SDC$^+$12]. These whole program analyzers are effective at finding bugs in whole programs, where every piece of code is known. Unfortunately, this means that whole program analyses are targeted at finding late bugs and not early bugs.

While they are not designed to be used in this way, whole program analyzers can be applied to try to find early bugs. Because whole programs assume knowledge of the whole program, it would be required to develop a driver program for the library that exercised all of the potential behavior of the library. This may be easier than developing a robust test suite because specific values need not be chosen. Instead, non-determinism can be used to select values. Such a driver is roughly equivalent to giving a logical specification for the library.

The problem with adapting the whole program analysis to library verification is that on almost all libraries, it will not work. As Figure 1.2 shows, existing whole-program analyses are designed to scale to analyzing whole dynamic language programs, not to handle the amount of non-determinism that occurs when creating a driver program to fully exercise a library. Non-determinism occurs in the heap, in attributes of objects, in values, and in functions. Handling this non-determinism, whether it comes from a logical specification or a driver program is the fundamental challenge of using static program analysis to eliminate bugs early in the development of dynamic language libraries.

1.3. Handling Non-deterministic Inputs to Libraries

In this dissertation, the goal is to support analysis of libraries, which means that support for unknowns at the inputs to library functions is required. Figure 1.2 shows the trade off that occurs. Scalability may be sacrificed to gain the ability to support non-deterministic unknown inputs precisely throughout an analysis.

To illustrate the problem of non-deterministic inputs to libraries, consider Figure 1.3. This shows a library function. This library function takes inputs in the form of objects in the heap such as o_1 and o_2 and functions such as f_1 and f_2. It then manipulates these objects, other connected objects in the heap, calls the functions and produces a new function f_3 and new object o_3. The problem with this type of library function is that takes not just simple values such as integers as input. It takes objects, functions, and the

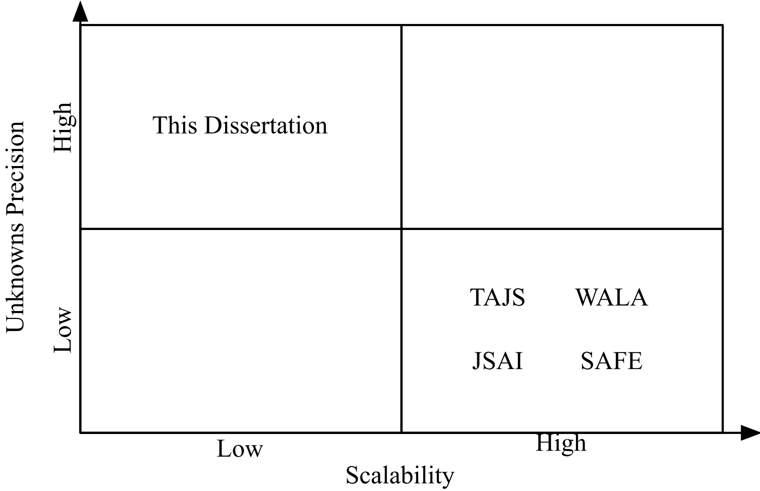


Figure 1.2. – Analysis landscape comparing scalability and unknown attribute name and function precision. Existing whole-program analyses are scalable, but do not effectively support unknowns as are used by libraries. This dissertation focuses on precise analyses for unknowns.

heap as input. Therefore analyzing a library by assuming the inputs are non-deterministic requires allowing functions, objects, and the heap to be non-deterministic.

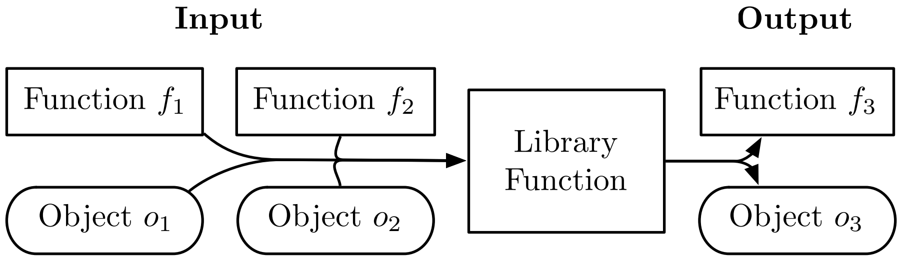


Figure 1.3. – A dynamic language library takes inputs of functions and objects and produces functions and objects as output.

Because inputs are non-deterministic and thus unknown, outputs must be specified relative to the inputs. For example, o_3 may be derived by combining parts of o_1 and o_2 whatever o_1 and o_2 may be. Therefore a specification is often *relational* between objects. Additionally, the specification may rely upon calling functions, such as f_1, and constructing new functions, such as f_3. Those functions that are being called may be unknown and may have side effects, so the specification must be able to represent the effects of function calls. Additionally, because the specification involves the heap, it is important that only the part of the heap that can be affected by the library need be specified. The portions of the heap that cannot be changed by the library should not affect the specification.

These aspects of the specification are also the key tenants of the analysis. If these assumptions are made at the specification level, the analysis must necessarily cope with these aspects or it is impossible for the analysis to validate a library with respect to its specification. Additionally the analysis must also be able to handle how the program arrives at the specification. For example, even if first-class attribute names are not required to specify the behavior of a library, if first-class attribute names are used to implement the library, analysis must support that feature or it cannot validate the library.

Abstraction The cornerstone of verification is abstraction. Abstraction allows considering an unbounded number of computations simultaneously and efficiently. In the case of non-determinism, all possible cases for a non-deterministic input could be considered simultaneously with an appropriate abstraction. Verification, whether it is with type systems, model checking [CES86, BR02, HJMS02], or abstract interpretation [CC77] is fundamentally about finding good abstractions. This dissertation is presented in the context of abstract interpretation. Abstract interpreters evaluate programs one step at a time representing many states of the program simultaneously using an abstract domain. An *abstract domain* implements a representation of an abstraction along with operations to manipulate the representation. In abstract interpretation, the analysis necessarily completes, finding an overapproximation of all realizable states in the program.

To develop an abstraction targeted at handling the non-determinism required for verifying a dynamic language library without a specific client, there are a number of necessary features:

- Native relational open object support — Objects need to support addition, removal, mutation, and iteration over attribute names. Additionally, when concrete values are unknown, open objects should be related to other open objects and variables from which they were constructed.

- Local heap abstraction — The heap is an important part of the input to a function. From one object, many other objects can potentially be manipulated by following pointers in the heap. The problem is that it is difficult to describe the entire heap going into and out of a library. There are many possible heaps created by client code and modeling all of them is problematic. Instead, a local heap abstraction requires that only the parts of the heap affected by or affecting the library be specified and manipulated. All other parts of the heap can be reasoned about compositionally.

- Strong updates — When too many unknown variables are in a program, as is often the case for libraries, where all inputs to the library are unknown, strong updates allow inferring better relations. Rather than only adding a new value as a possibility to an abstraction, a *strong update* removes the old value before adding the new one to keep better precision. Critically, strong updates are needed in loops that manipulate objects to be able to prove properties such as object copying.

- Unknown function abstraction — Dynamic language libraries often take functions as arguments or have functions reachable in the heap from arguments that are then called by the library. Developers construct their libraries so that when they accept functions as input that there are few (if any) requirements on those functions. Analysis of libraries should work in a similar way. If a function makes no assumptions, no assumptions should be needed to complete the analysis.
- Flexible abstractions — Different programs have different needs. Abstractions should be composable so that they can be used in as many different situations as possible. This allows for abstractions to be tuned for certain libraries. This tuning allows improved efficiency while allowing the extension of an analysis to support new libraries easier.

What unifies all of the various features is the concept of a heap for a dynamic language. Native open object abstractions relate objects in the heap to other objects and values in the heap. Strong updates can be achieved by ensuring that objects in the heap that are being modified are always single, non-summary objects and attributes that are being modified are always single, non-summary attributes. Function abstractions are about determining the portion of the heap that can be affected by a function and how it can be affected. By parameterizing a heap abstraction by other abstractions, a heap abstraction can have the flexibility to work in a wide variety of situations. In short, all of the features are closely tied to the heap abstraction.

1.4. Thesis Statement

Because dynamic language libraries manipulate and use objects in the heap and because they call and construct functions with side-effects, there is a need for local heap reasoning along with a way of relating objects to one another in that heap. The use of relation abstractions for sets provides an answer:

The combination of local heap reasoning with sets provides a means to construct direct, parametric abstractions suitable for automatically analyzing dynamic language libraries.

A need for local heap reasoning suggests the use of separation-logic-based heap abstractions [IO01, Rey02, BCO05, CR08]. These abstractions are designed to allow local reasoning about heaps with inductively defined data structures through (sometimes parametric) inductive predicates. Instead of relying on inductively defined data structures, dynamic languages programs and libraries typically rely on their objects to represent unbounded data structures.

Uniquely, this dissertation centers a new separation-logic-based heap abstraction on an abstraction for sets. These sets are used to relate sets of attribute names to other

sets of attribute names in other objects. They are used to define materialization and summarization in the heap, to enable strong updates in both the heap and objects. They are also used to relate values to other values.

Because of the fundamental dependence of dynamic language programs and libraries on the extensibility of object objects, the resulting set-oriented, separation-logic-based heap abstraction accomplishes precise abstractions in a wide range of dynamic language libraries. By using a relational set abstraction, attributes of unknown objects passed to library functions can be related to objects generated by the library function. Consequently, even when objects are completely unknown, sets allow directly abstracting open objects by using relations between objects.

To define this set-oriented, separation-logic-based heap abstraction, this dissertation not only defines the separation-logic-based heap, but also defines a family of suitable, parametric relational set domains. It also presents techniques based on the heap abstraction to abstract unknown functions. As a result, this dissertation defines and demonstrates techniques suitable for automatically verifying object-manipulating library routines.

The goal of this dissertation is to develop local heap reasoning abstractions such that when combined with precise set abstractions address many of the key challenges of analyzing libraries. Ideally, they would be able to fully precisely analyze dynamic language libraries. In practice, they can be used to verify developer-specified assertions and pre/postconditions for dynamic language libraries when common dynamic language development idioms are used, such as object copying, object filtering, and calling user-supplied callbacks.

1.5. Summary

This dissertation describes four advances in the handling of unknowns that arise in the analysis of dynamic language libraries. In Chapter 2 the need for analyzing dynamic language libraries is motivated with code examples. Chapter 3 introduces by example the four contributions that are shown in Figure 1.4. These contributions form three shown abstractions that work together to make an analysis for dynamic language libraries.

Chapter 4 presents the heap with open objects (HOO) abstraction, which is the key analyzing programs with unknown open object manipulation. It is an abstraction for dynamic language heaps that includes support for open (extensible) objects. However, it is parameterized by an abstraction for sets. This abstraction for sets controls the possible values stored in the heap as well as the precision of the abstraction of those values. Chapter 4 also presents attribute/value trackers, which extend object abstractions like HOO to increase precision when analyzing programs that copy objects by providing a form of parametric polymorphism.

Chapter 5 presents desynchronized separation, which is a means to analyze functions that call unknown functions. Since side effects in the heap are the biggest concern when calling unknown functions, it is a parametric abstraction for the heap. It takes an abstraction for heaps and transforms it into an abstraction for heaps where calls to unknown functions are possible.

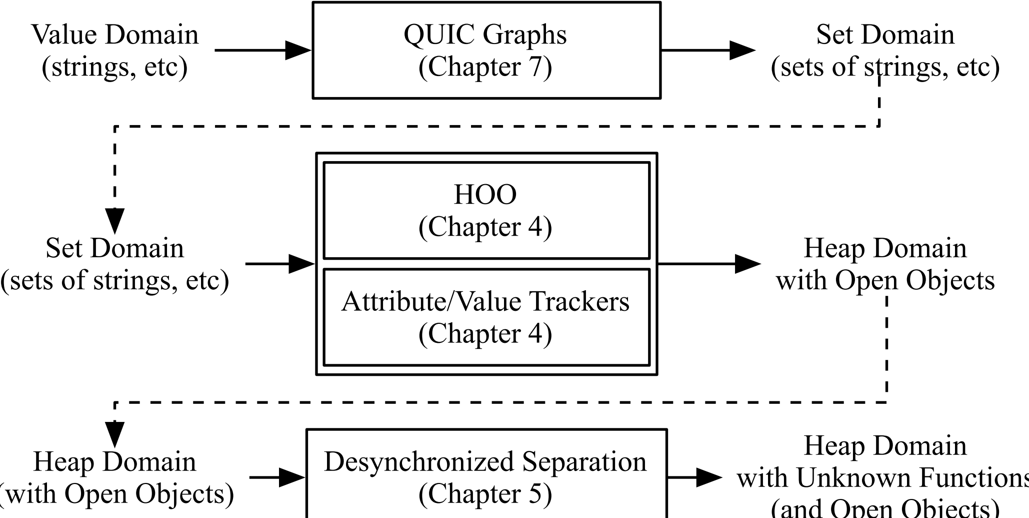


Figure 1.4. – Abstractions introduced in this dissertation are parametric. The QUIC graphs abstraction converts any abstraction for values (such as strings, integers, etc.) into an abstraction for sets of values. The HOO abstraction (with and without attribute/value trackers) converts an abstraction for sets representing addresses, attribute names, and values into an abstraction for a dynamic language heap. Desynchronized separation converts an abstraction for heaps into an abstraction for heaps with calls to unknown functions. The dashed lines show how these abstractions work together for the implementation presented in this dissertation.

Chapter 6 combines the heap abstraction with the unknown function abstraction and discusses the result of applying the analysis to dynamic language libraries. It combines the abstractions via the dashed lines shown in Figure 1.4 building upon an abstraction for sets that is described in Chapter 7. The application of the combined analysis is to demonstrate the combination of the abstractions in analyzing a number of JavaScript-inspired functions, many of which are extracted from libraries such as Traits.js and Google Closure.

Chapter 7 presents the QUIC graphs abstraction — a flexible, parametric abstraction for sets that is used by HOO in the combined analysis. QUIC graphs take any abstraction for values, such as strings or integers, and transform that into an abstraction for sets of strings or sets of integers. In addition to its use as part of the HOO-based analysis, it is evaluated separately on set manipulating python programs.

2. Motivation: JavaScript Library Verification without Client Code

This chapter motivates the need for various kinds of analysis with concrete code examples. To do this, it tells the story of a hypothetical JavaScript developer Jennifer who is an experienced JavaScript developer who wants to improve not only her own development practices, but also other JavaScript developers'. Right now, the best way she can think of improving her practice is to create a class system for JavaScript. JavaScript does not have classes built in and everybody she works with knows how to use classes, so it would be easier if JavaScript had classes.

Rather than creating a new language with classes that compiles to JavaScript, Jennifer realizes that using built-in syntax to JavaScript, she can make something satisfactory that looks enough like a native class implementation that it will be acceptable. To demonstrate the idea to her peers, she writes the simple example shown in Figure 2.1. This use of the class implementation makes a class for storing two-dimensional points.

```
var Point = Class({
  // member variables
  x: 0,
  y: 0,
  // constructor
  init: function(x,y) {
    if(x) this.x = x;
    if(y) this.y = y;
  },
  // methods
  mag: function() {
    return Math.sqrt(this.x * this.x +
                     this.y * this.y);
  }
});
```

Figure 2.1. – The class `Point` looks similar to a class in a language like Java, but it is a JavaScript function that is created by calling the `Class` function on a configuration object.

From a JavaScript perspective the code is a call to a function called `Class` that takes a single parameter. This parameter is a configuration object that determines what the resulting class will be. Here, it has attributes 'x', 'y', 'init', and 'mag', which are associated

with initial values. There is no formal distinction between a member and a method. A member is a non-function value and a method is a function value. The binding of ***this*** happens automatically through JavaScript's object system, so functions and methods can be the same. The one special value here is 'init', which is the constructor for the class and should not appear in the instances because it is not simply a function.

The way this code is written admits some common JavaScript idioms. For example, because 'init' takes two written parameters, but JavaScript does not enforce this, it checks the value of the parameters and if they are provided (and not falsy) they are used, otherwise the default values are kept. This allows the constructor to be called in several ways.

```
// defaults kept
var pt_origin = Point();
// default value for y
var pt_x5 = Point(5);
// default value for x
var pt_y6 = Point(false,6);
// fully specified point
var pt_x2_y3 = Point(2,3);
// print the distance from the origin
print(pt_x2_y3.mag());
// print coordinates for pt_y6
print( pt_y6.x + "," + pt_y6.y );
```

Figure 2.2. – Construction of several instances of `Point` using the class system.

To help understand how classes created with `Class` might be used, Figure 2.2 shows the instantiation and use of several `Point` objects. First, there is, in effect, a default constructor. Both parameters to the constructor function are absent and thus are replaced with the value `undefined`, a special value that is the result when reading absent object fields and elided function parameters. Because `undefined` is *falsy*, it behaves as false when used in a Boolean context, such as the test of an if statement. Consequently, in the constructor both if tests fail and the default values for `x` and `y` are used. Similarly, when constructing `pt_x5`, the parameter `y` is not provided and is thus `undefined`. Therefore `y` is assigned the default value. The `pt_y6` example is a bit different because it relies upon an explicit falsy value: **`false`**. It is used to select the default value for `x`, while initializing `y` to 6. Finally, `Point` can be constructed with a fully specified parameter list.

The result of calling a class constructor is an object and behaves as such. A call to the `mag` method, because it is called as a field of an object (in this case, using the dot notation), automatically binds ***this*** to the object, which in this case is `pt_x2_y3`. Consequently, when inside the method, the use of ***this*** to look up `x` and `y` behaves as if looking up `x` and `y` in `pt_x2_y3`. Similarly, members `x` and `y` are directly accessible.

Given that this looks sufficiently close a class definition in other languages, Jennifer

decides to see if she can design a reusable library in JavaScript that implements this functionality. Like nearly everything in JavaScript, it will rely on JavaScript's flexible object system, combined with first-class functions and closures.

2.1. Class Library Requirements

To ensure that the `Class` function is correctly implemented, Jennifer notes the following list of requirements:

- Classes should be immutable. After a class is constructed, the definition should not be allowed to change. This means that two objects instantiated from the same class using the same parameters should result in identical objects regardless of when those two objects are constructed.
- Objects instantiated from classes should be derived from the class. Of course, because JavaScript is a dynamic language, the `init` function is free to add or remove attributes having started with those created from the class. This means that prior to calling `init`, attributes and values of the class should match exactly the attributes and values initially used to construct the class. However, because `init` is a constructor, it should not be a method of the resulting class.

2.2. Implementing the JavaScript Class Library

The code for the resulting `Class` is shown in Figure 2.3. It transforms a configuration object `cfg` into a function that when called creates an instance of the class. The `Class` function consists of three parts: (1) the `copy` function that is responsible for making shallow partial copies of objects; (2) the protected backup into variables `attrs` and `init`; and (3) the class constructor function `ctor`, which initializes the instance object before running the client-supplied constructor `init`.

The `copy` function is responsible for partially copying objects that are passed to it. It takes three objects as parameters `res`, `src`, and `exc`. The `src` parameter is the source object for the copy. Each attribute and corresponding value of the `src` object will be considered for copy. The `exc` parameter lists all of the exceptions to the copy. Each attribute in `exc` will be skipped during the copy. The copy process is shown in Figure 2.4, demonstrating the copying that occurs in the class constructor during the construction of a `Point` object. In the code, there is a **`for-in`** loop that iterates over each attribute in the `src` object. Because the iteration order is unspecified, Figure 2.4 shows one possible order of events. First, ①, the attribute 'y' is compared with the attributes in `exc`. Because it is not in the attributes of `exc` it is copied into `result`. Then, this process is repeated for 'mag', ②, and 'x', ③. Finally, when it gets to 'init', ④, it finds that the attribute is in `exc` and thus it is not copied.

The protected backup is responsible for ensuring that the class is immutable. By default, all accessible objects in JavaScript are mutable. Consequently, it is possible that the definition of a class could change over time if it is not in some way made inaccessible.

```
          var Class = function(cfg) {
        ⎧   function copy (res,src,exc) {
        ⎪     for(var p in src)
copy    ⎨       if(!(p in exc))
        ⎪         res[p] = src[p];
        ⎪     return res;
        ⎩   }
protected ⎧ var attrs = copy({},cfg,{});
backup    ⎩ var init = cfg.init;
        ⎧   function ctor() {
        ⎪     var result = copy({},attrs,{init:""});
constructor ⎨ init.apply(result, arguments);
        ⎪     var rv = result;
        ⎩   }
            return ctor;
          }
```

Figure 2.3. – Class implementation using open objects, first-class functions and closures.

This code implements such protection. A local copy of the `cfg` object is created and a local copy of `init` is created for easier access. Because these are local variables, they are inaccessible globally, and thus cannot be externally mutated.

The class constructor is responsible for first, creating the object that is the instance of the class, second, initializing the object with various attributes and default values, and third, delegating to the `init` function, which is the client-supplied constructor. The creation of the object allocates a new empty object on the heap, which is immediately passed to `copy` for initialization. The `copy` function performs initialization by copying from the protected backup (not the `cfg` object). This automatically initializes all members to their default values and adds all methods to the object. Since the client-supplied constructor is not a normal method, it should not be added to the resulting object and thus is excluded from the `copy` operation. Finally, using the JavaScript indirect function call syntax, the client-supplied constructor is called, passing it the newly constructed object as ***this***. Note that regardless of the value that the client-supplied constructor returns, the newly allocated object is returned at the end of instance construction.

Now that the library has been created, Jennifer wants to deploy the library so that others can start using it. But Jennifer is savvy. She knows that it could be a costly error to release the library without making sure that it works. She can run tests like the example in Figure 2.2 and check that the resulting values are correct, but she wants to know that in the future this code will not go wrong — that she will not be responsible for the problems of other developers. To do this she is going to use static analysis.

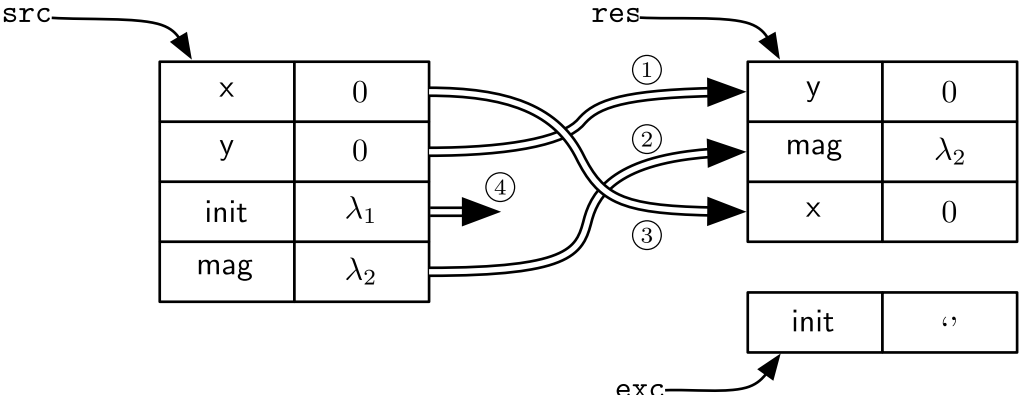


Figure 2.4. – Iteratively copying the `attrs` object to the `result` object, excluding attributes that are also in the `exc` object.

2.3. Analyzing the JavaScript Class Library

To gain assurance that the class library behaves as it should, Jennifer converts the requirements of the class library into more specific intentions in the form of properties. These properties can then be encoded as annotations in the program to inform the static analysis of the intended correct behavior of the code:

1. The protected backup contains a full copy of the configuration object `cfg`, no matter which attributes and values are present in the configuration object. This relies on `copy` behaving correctly and making a full copy of `cfg`.

2. In the call to the generated `ctor` function, before the call to `init`, the `result` object has a copy of all attributes and values from the protected backup except for `init`. The `init` attribute is the client-supplied constructor and should not be present in the result.

3. The result of the call to the generated `ctor` function should be the `result` object whatever the call to `init` may do. The contents of the `result` object depend on the call to `init`.

4. After completing the call to `Class`, the protected backup is unreachable from the global object, the `cfg` object, or the result object. This indicates that the protected backup is indeed protected. If this does not hold, the `Class` is actually mutable.

5. The call to `init` should neither change the protected backup, nor provide a means for any other code to change the protected backup. This ensures that no matter how many instances are created from a single class, the class is always immutable.

These properties express the expectations that Jennifer has for the class library. In short, despite the fact that the class system is created from mutable objects with functions and closures, the class system should behave like a class system. Classes should be

immutable and should produce instance objects that are derived from the configuration object at the time the class was constructed.

Jennifer needs a static analysis that can prove the above properties. The correctness of Property 1, the protected backup, is dependent on a native open object abstraction, strong updates, and a relational heap abstraction. This is because the `copy` function does not know what the `cfg` object has for attributes and values and yet the property must determine if an equality relationship exists between the original `cfg` and the protected backup. Property 2, the initialization of `result`, is similar, but more challenging. Not only must the analysis infer that a copy occurred, but it must also infer that the `init` attribute is not in the result. Property 3, the result of calling `init`, must reason about unknown functions. Whatever effects this call may have on reachable memory, the property should hold. This requires a combination of local reasoning and an abstraction for the behavior of unknown functions. Property 4, the unreachability of the protected backup, requires only a simple heap abstraction since local variables are always unreachable and `attrs` is never stored in any reachable objects. Simlarly, Property 5, the immutable protected backup, requires only a simple heap abstraction as these variables are not written and not transferred to reachable objects. The challenge with this property is reaching it when a call to an unknown function is called previously. Otherwise, these last two properties can both be proven with existing whole program analyses.

3. Overview: Analysis of JavaScript Libraries

This chapter introduces by example the analyses presented formally, in detail throughout the rest of the dissertation. As an example, this chapter makes use of the `Class` function that was detailed in Chapter 2. However, because of the focus on open objects, in Section 3.1 I define the open-object-focused JavaScript variant that restricts the JavaScript syntax to statements pertinent to open object manipulation. Then, in Section 3.2, I show how common JavaScript idioms can be represented in open-object-focused JavaScript. In Section 3.3, I introduce the graphical representation used throughout the paper to represent abstract states. Finally, in Section 3.4, I present an example analysis of `Class` that demonstrates the three main challenges addressed by the analyses in this dissertation: unknown attribute access, iteration over objects, and calls to unknown functions.

3.1. Open-Object-Focused JavaScript Language

JavaScript is a complex language. It is very much a dynamic language as typing is dynamic, objects can be directly manipulated, lexical scoping can be dynamically manipulated, and strings can be converted to code and back. All of these features make it exceedingly difficult to specify a semantics [GMS12, BCF$^+$14], let alone design a static analysis. In reality, developers do not commonly abuse [RLBV10] these features of JavaScript and tend to use them in relatively structured ways. In fact, popular tools such as JSLint[1] actively discourage the use of many of the dynamic features.

To focus the dissertation on the core problems introduced by libraries such as `Class`, where the primary operations are reading from, writing to, and iterating over unknown or partially unknown objects along with calling unknown functions, I introduce open-object-focused JavaScript. The syntax of the language is shown in Figure 3.1. The language is defined as commands k, where each k can manipulate program variables `x`, `y`, `z`. Additionally, there are special program variables that represent important information for functions: `res` for the result of the functions, `glbl` for the global object, `this` for the object that is bound to this, `clos` for an object that represents variables in the closure, and `args` for an object representing the arguments to a function.

The language is intended to be like JavaScript with respect to objects, but simplified in several ways. Conditions are restricted to those that have to do with object membership. The program is in a-normal [SF92] form, where all commands only accept variables as operands and not nested expressions. Functions are closure converted and lowered into

[1] http://www.jslint.com

```
k  ::=  x = c
     |  x = y
     |  x = {}
     |  x = y[z]
     |  x[y] = z
     |  for(x in y) { k }
     |  if(x in y) { k1 } else { k2 }
     |  k1; k2
     |  x = function(res, glbl, this, clos, args) { k }
     |  x(y1, y2, y3, y4, y5)
```

Figure 3.1. – Open-Object-Focused JavaScript Language

a specific form with the five special arguments as described previously. This closure-converted form allows most of the behavior of JavaScript, but it makes explicit much of the behavior of function declaration and application so that internal special variables can be referenced during analysis.

To define the semantics of this language, I first define four helpers for association lists. These are shown in Figure 3.2. Association lists consist of the empty list [] and a constructor :: that builds a list from an element and another list. Each element of an association list is a pair of key and a corresponding value. I use the symbol r to represent an arbitrary association list and σ to represent a heap, which is a nesting of association lists. The symbols used for keys and values depend on the type of the information stored in the association list. Here a, b, and c are keys or values and v is a value. The first operation $\mathrm{Dom}\,(r) = d$ gets the domain of an association list by collecting a set d of all of the keys of the association list starting from r. The second operation $\mathrm{Lookup}\,(a, r) = v$ finds a value v that corresponds to a particular key a. If the key is not in the association list, no value v can be retrieved and no proof tree can be constructed. The third operation $\mathrm{Locals}\,(\sigma, \sigma') = \sigma''$ extracts the local variables from the heap σ and adds them to σ' giving σ'' and similarly, the fourth operation $\mathrm{Heap}\,(\sigma) = \sigma'$ extracts everything that is not a variable from σ, leaving only objects in the result σ'. Finally, the special value () is defined as the unit value. It is used to represent pointers in the heap. An object that has the special unit value as an attribute can be treated as a basic pointer. Consequently, no explicit stack or environment is required in the semantics. Only the helper syntax $x \mapsto v$ is used to simplify the presentation of the semantics.

The semantics of each command in the language are given in Figures 3.3 and 3.4. The semantics are defined as big-step operational semantics with a judgment of the form $\langle\sigma\rangle\, k\, \langle\sigma'\rangle$, which evaluates a command k on a heap σ, producing a heap σ'. Program variables x, y, or z are assumed to be placeholders for object addresses, and thus no environment is necessary. A heap is an association list that maps each address a to an object o. An object o is an association list that maps each attribute (field, property, etc.) p to a value v. A value v can be an address a, a constant such as a string constant, a command k (representing a function body), or the special undefined value undef.

$$\boxed{\mathrm{Dom}\,(r) = d}$$

$$\frac{}{\mathrm{Dom}\,([]) = \emptyset}$$

$$\frac{\mathrm{Dom}\,(r) = d}{\mathrm{Dom}\,((a, b) :: r) = a \cup d}$$

$$\boxed{\mathrm{Lookup}\,(a, r) = v}$$

$$\frac{}{\mathrm{Lookup}\,(a, (a, b) :: r) = b}$$

$$\frac{a \neq b \qquad \mathrm{Lookup}\,(a, r) = v}{\mathrm{Lookup}\,(a, (b, c) :: r) = v}$$

$$\boxed{\mathrm{Locals}\,(\sigma, \sigma') = \sigma''}$$

$$\frac{}{\mathrm{Locals}\,([], \sigma') = \sigma'}$$

$$\frac{\mathrm{Locals}\,(\sigma, \sigma') = \sigma''}{\mathrm{Locals}\,((x \mapsto v) :: \sigma, \sigma') = (x \mapsto v) :: \sigma''}$$

$$\frac{\mathrm{Locals}\,(\sigma, \sigma') = \sigma'' \qquad r \neq (x \mapsto v)}{\mathrm{Locals}\,(r :: \sigma, \sigma') = \sigma''}$$

$$\boxed{\mathrm{Heap}\,(\sigma) = \sigma'}$$

$$\frac{}{\mathrm{Heap}\,([]) = []}$$

$$\frac{\mathrm{Heap}\,(\sigma) = \sigma'}{\mathrm{Heap}\,((x \mapsto v) :: \sigma) = \sigma'}$$

$$\frac{\mathrm{Heap}\,(\sigma) = \sigma' \qquad r \neq (x \mapsto v)}{\mathrm{Heap}\,(r :: \sigma) = r :: \sigma'}$$

$$(x \mapsto v) :: \sigma \stackrel{\text{def}}{=} (x, ((), v) :: []) :: \sigma$$

$$(x \mapsto v) \in \sigma \stackrel{\text{def}}{=} \mathrm{Lookup}\,(x, \sigma) = ((), v) :: []$$

Figure 3.2. – Helper definitions for Open-Object-Focused JavaScript define the domain and a lookup operation for association lists

$$\boxed{\langle\sigma\rangle\, k\, \langle\sigma'\rangle}$$

$$\frac{}{\langle\sigma\rangle\ \mathtt{x\ =\ }c\ \langle(\mathtt{x}\mapsto c) :: \sigma\rangle}\ \textsc{C-Cst} \qquad \frac{(\mathtt{y}\mapsto v)\in\sigma}{\langle\sigma\rangle\ \mathtt{x\ =\ y}\ \langle(\mathtt{x}\mapsto v) :: \sigma\rangle}\ \textsc{C-Var}$$

$$\frac{\mathrm{Dom}\,(\sigma) = d \qquad a\notin d}{\langle\sigma\rangle\ \mathtt{x\ =\ \{\}}\ \langle(x\mapsto a) :: (a, []) :: \sigma\rangle}\ \textsc{C-Alloc}$$

$$\frac{(\mathtt{y}\mapsto a)\in\sigma \qquad (\mathtt{z}\mapsto p)\in\sigma \qquad \mathrm{Lookup}\,(a,\sigma) = o \qquad \mathrm{Lookup}\,(p,o) = v}{\langle\sigma\rangle\ \mathtt{x\ =\ y[z]}\ \langle(\mathtt{x}\mapsto v) :: \sigma\rangle}\ \textsc{C-Ld-P}$$

$$\frac{(\mathtt{y}\mapsto a)\in\sigma \qquad (\mathtt{z}\mapsto p)\in\sigma \qquad \mathrm{Lookup}\,(a,\sigma) = o \qquad \mathrm{Dom}\,(o) = d \qquad p\notin d}{\langle\sigma\rangle\ \mathtt{x\ =\ y[z]}\ \langle(\mathtt{x}\mapsto \mathsf{undef}) :: \sigma\rangle}\ \textsc{C-Ld-N}$$

$$\frac{(\mathtt{x}\mapsto a)\in\sigma \qquad (\mathtt{y}\mapsto p)\in\sigma \qquad \mathrm{Lookup}\,(a,\sigma) = o \qquad \mathrm{Lookup}\,(\mathtt{z},s) = v}{\langle\sigma\rangle\ \mathtt{x[y]\ =\ z}\ \langle(a, (p,v) :: o) :: \sigma\rangle}\ \textsc{C-St}$$

$$\frac{\begin{array}{c}(\mathtt{x}\mapsto v)\in\sigma\\ (\mathtt{y}\mapsto a)\in\sigma \qquad \mathrm{Lookup}\,(a,\sigma) = o \qquad \mathrm{Dom}\,(o) = d \qquad v\in d \qquad \langle\sigma\rangle\, k_1\, \langle\sigma'\rangle\end{array}}{\langle\sigma\rangle\ \mathbf{if}\mathtt{(x\ }\mathbf{in}\mathtt{\ y)\ \{\ }k_1\mathtt{\ \}\ }\mathbf{else}\mathtt{\ \{\ }k_2\mathtt{\ \}}\ \langle\sigma'\rangle}\ \textsc{C-If-P}$$

$$\frac{\begin{array}{c}(\mathtt{x}\mapsto v)\in\sigma\\ (\mathtt{y}\mapsto a)\in\sigma \qquad \mathrm{Lookup}\,(a,\sigma) = o \qquad \mathrm{Dom}\,(o) = d \qquad v\notin d \qquad \langle\sigma\rangle\, k_2\, \langle\sigma'\rangle\end{array}}{\langle\sigma\rangle\ \mathbf{if}\mathtt{(x\ }\mathbf{in}\mathtt{\ y)\ \{\ }k_1\mathtt{\ \}\ }\mathbf{else}\mathtt{\ \{\ }k_2\mathtt{\ \}}\ \langle\sigma'\rangle}\ \textsc{C-If-N}$$

$$\frac{\langle\sigma\rangle\, k_1\, \langle\sigma'\rangle \qquad \langle\sigma'\rangle\, k_2\, \langle\sigma''\rangle}{\langle\sigma\rangle\, k_1\mathtt{;}\ k_2\, \langle\sigma''\rangle}\ \textsc{C-Seq}$$

$$\frac{\begin{array}{c}(\mathtt{x}\mapsto k)\in\sigma \qquad (\mathtt{y}_1\mapsto v_1)\in\sigma \qquad (\mathtt{y}_2\mapsto v_2)\in\sigma \qquad (\mathtt{y}_3\mapsto v_3)\in\sigma\\ (\mathtt{y}_4\mapsto v_4)\in\sigma \qquad (\mathtt{y}_5\mapsto v_5)\in\sigma \qquad \sigma_f = (\mathtt{clos}\mapsto v_4) :: (\mathtt{args}\mapsto v_5) :: \sigma'\\ \langle(\mathtt{res}\mapsto v_1) :: (\mathtt{glbl}\mapsto v_2) :: (\mathtt{this}\mapsto v_3) :: \sigma_f\rangle\, k\, \langle\sigma''\rangle\\ \mathrm{Locals}\,(\sigma,\sigma'') = \sigma''' \qquad \mathrm{Heap}\,(\sigma) = \sigma'\end{array}}{\langle\sigma\rangle\ \mathtt{x(y1,\ y2,\ y3,\ y4,\ y5)}\ \langle\sigma'''\rangle}\ \textsc{C-App}$$

$$\frac{}{\begin{array}{c}\langle\sigma\rangle\\ \mathtt{x\ =\ }\mathbf{function}\mathtt{(res,\ glbl,\ this,\ clos,\ args)\ \{\ }k\mathtt{\ \}}\\ \langle(\mathtt{x}\mapsto k) :: \sigma\rangle\end{array}}\ \textsc{C-Fun}$$

Figure 3.3. – Semantics of open-object-focused JavaScript

$$\boxed{\vdash \langle\sigma\rangle\,(\text{x in } d)\ k\,\langle\sigma'\rangle}$$

$$\frac{}{\vdash \langle\sigma\rangle\,(\text{x in } \emptyset)\ k\,\langle\sigma\rangle}\ \text{C-For-N}$$

$$\frac{\langle(\text{x}, v) :: s, h\rangle\, k\,\langle\sigma'\rangle \qquad \vdash \langle\sigma'\rangle\,(\text{x in } d)\ k\,\langle\sigma''\rangle}{\vdash \langle\sigma\rangle\,(\text{x in } \{v\} \uplus d)\ k\,\langle\sigma''\rangle}\ \text{C-For-P}$$

$$\frac{(\text{y} \mapsto a) \in \sigma \qquad \text{Lookup}\,(a, \sigma) = o \qquad \text{Dom}\,(o) = d \qquad \vdash \langle\sigma\rangle\,(\text{x in } d)\ k\,\langle\sigma'\rangle}{\langle\sigma\rangle\,\texttt{for(x in y) \{ } k \texttt{ \}}\,\langle\sigma'\rangle}\ \text{C-For}$$

Figure 3.4. – Semantics of open-object-focused JavaScript **`for`** command

Command Semantics The first command `x = c` assigns a constant c, including undef or any k to the variable x by replacing the heap with a new version of the heap where the program variable x maps to the constant c. This ensures that x is now included in the $\text{Dom}\,(\sigma')$ and that $(\text{x} \mapsto c) \in \sigma'$. This leaves the other objects in the heap unmodified, but effectively declares a variable that exists until the end of the function scope.

The second command `x = y` assigns the value read from y to x. First, it looks up the value v from the heap σ using a form of Lookup () and then it binds that value to x in the same way as assigning a constant. Any value can be copied from one variable to another in this way.

The command `x = {}` allocates a new object and assigns it to x. Allocating a new object requires that the object exists at an address a that is distinct from any address already allocated. To ensure that this is the case, a is constrained to be distinct from any address already in the domain d of the heap. In the result, the heap is extended twice: once for the new empty object to the heap and once for binding the newly allocated address to x.

The load command `x = y[z]` is broken up into two rules. The first rule applies if the particular attribute from variable z exists inside the object referenced by y. The second applies if this is not the case. In either case, the address a is retrieved from the heap and then the corresponding object o is retrieved. If the attribute p, which came from y, is in the domain of the object o, the C-Ld-P rule applies. Otherwise, the C-Ld-N rule applies. The difference is that if the attribute is present, the corresponding value v is bound to x. Otherwise the value undef is bound to x. Note that this load behavior differs from JavaScript in that there is no prototype chain that is automatically followed. Any use of the prototype chain must be explicitly written in the code.

The store command `x[y] = z` mirrors closely that of the JavaScript store. It finds the appropriate object o, and attribute p as in the load case and the appropriate value v and then adds an updated version of the object with p bound to v in that object to the heap. This effectively overwrites the previous object on the heap at address a. This

is different from JavaScript only in that it does no implicit conversion of the attribute. Attributes must always be strings.

The **if** command branches depending on the presence of a particular attribute in an object. If the attribute is present, the C-If-P rule applies and the first case command k_1 is evaluated. Otherwise the C-If-N rule applies and the second case command k_2 is evaluated.

The sequencing of commands with the C-Seq rule threads the heap from one command to another.

The declaration and application of functions through the C-Fun and C-App rules use commands as values. Because all functions have the same signature taking five arguments respectively representing the result, the global object, the ***this*** object, the closure, and the arguments object, only the body of the function is important. Therefore, the body of the function is used as a value and is written to variable x. Function application reverses this process by looking up all of the arguments in the heap σ and then extending the heap with each of the parameter variables bound to the respective values. The body of the function is also looked up, giving a command k, which then can be evaluated on the extended heap.

The final operation of the language, shown in Figure 3.4, is the **for-in** loop. This loop iterates over all attributes of an object, binding them to a given variable. To give the semantics of this command, I introduce a new judgment that represents a partially completed loop. The $\vdash \langle\sigma\rangle\ (\mathrm{x}\ \text{in}\ d)\ k\ \langle\sigma'\rangle$ takes a step in the loop over a set of attributes d for an object. The rule C-For-N applies if the set d is empty. In this case, the iteration is complete and the same heap is in the result of the evaluation. In the other case, C-For-P applies and an element v is selected from the set and bound to x before evaluating the body of the loop. Upon completion of the body, the next iteration of the loop continues.

The **for-in** differs from the JavaScript equivalent in several ways. First, it selects the set of attributes for iteration once when the iteration starts. JavaScript supports modification of the object that is being iterated over and may visit newly added attributes. Additionally, JavaScript, by default, iterates over not only the provided object, but also every object in the prototype chain.

I will use this open-object-focused JavaScript for the remainder of the dissertation with the exception that I will not always present an a-normalized version of the code. The a-normalization process is simple and thus will only be presented when interesting for a particular analysis. Additionally, I will apply simplifications to the converted code eliminating unused elements. For example, functions that do not need a closure, will not construct one. Lastly, I will use array notation to represent the construction of an object with string attributes representing increasing numbers starting from '0'.

Extending to Full JavaScript While open-object focused JavaScript is simplified compared to regular JavaScript, it possesses much of the complexity. By adding simple expressions that define other JavaScript operations such as type coercions, number and string operations, and standard library operations, via a standard desugaring processes [GSK10], it is possible to reduce full JavaScript down to a core calculus similar

to this. For example, the addition operation may add two numbers or concatenate two strings. To determine which is being performed, the types of the two operands are checked. All of this can be made explicit through desugaring. Similarly, full support for operations such as computed attributes can be added through a simple language extension.

3.2. Real-World Open-Object-Focused JavaScript

While open-object-focused JavaScript is not identical to real-world JavaScript, many real world programs still work in this language. For example, the `Class` function detailed in Section 2 is entirely representable in open-object-focused JavaScript. Additionally, most libraries that offer features similar to `Class`, such as implementations of traits and mixins fall into open-object-focused JavaScript. As a result, it is possible to consider analyzing real JavaScript libraries that have been translated into open-object-focused JavaScript by some simple preprocessing.

A version of `Class` converted to open-object-focused JavaScript is shown in Figure 3.5. The same three parts exist as before: the copy function, the protected backup, and the constructor function. The primary difference is that the functions have been closure converted. This means that any variables that are non-local to the function are accessed through a closure object called `clos`. This section gives a brief tour of the operations that are performed to transform JavaScript to open-object-focused JavaScript.

The resulting `Class` now has functions that always take five parameters. In JavaScript functions are all variadic — the number of parameters is not fixed, so in this conversion, that variadic list of parameters is converted into an object with successive numbers for the attributes that correspond to the values. This is similar to how C uses the `argv` array to represent a variable number of arguments on the command line. Consequently, all accesses to function arguments are replaced with access to the appropriate index of the object.

Additionally, the global object, represented by `glbl` is an object that is accessible from any function in JavaScript. To model this without having global variables explicitly as part of the program, I thread the global object through function calls. Therefore, functions can all read from, write to, or otherwise access and modify the global object by naming it explicitly. If the program is truly lexically scoped, determining if the global object is being used is simply a matter of determining if the variable being accessed is declared in any enclosing scope. In this code, `glbl` is not used, as there are no global variables.

Similarly, the ***this*** object is explicitly named. JavaScript implicitly creates an object called ***this*** within every function. The ***this*** object is the object to which a function is bound, so that if a method is called on an object, the object on which it was called is passed as ***this***. By explicitly naming it, preprocessing causes the application of the `init` function in the constructor to have the appropriate use of ***this*** for the result object.

The `res` argument is an object where the return value can be written. By convention, the return value is written to the 'res' attribute. By making the result also a parameter,

```
Class = function(res, glbl, this, clos, args) {
  // define the copy function
  copy = function(res, glbl, this, clos, args) {
    res = args[0];
    src = args[1];
    exc = args[2];
    for(p in src) {
      if(p in exc) { } else {
        res[p] = src[p];
      }
    }
  };

  // make the protected backup
  attrs = {};
  copy({}, glbl, glbl, {}, [attrs,args[0],{}]);
  init = args[0]["init"];

  // create a closure object for the constructor
  clos = {};
  clos["attrs"] = attrs;
  clos["init"] = init;
  clos["copy"] = copy;

  // define the constructor body
  fun = function(res, glbl, this, clos, args) {
    result = {};
    exc = {};
    exc["init"] = "";
    clos["copy"]({}, glbl, glbl,
                 {}, [result,clos["attrs"],exc]);
    clos["init"]["fun"]({}, glbl, result,
                        clos["init"]["clos"], args);
    res["res"] = result;
  }

  // build the constructor object
  ctor = {};
  ctor["fun"] = fun;
  ctor["clos"] = clos;

  res["res"] = ctor;
}
```

Figure 3.5. – Open-object-focused JavaScript translation of `Class` makes explicit many of the basic JavaScript operations

the analysis need not consider return values, which simplifies the discussion. In this particular code, there is no need for the use of return values and thus they are never accessed, but they will be described in the specifications of these functions in the next section.

3.3. Abstraction: Representation of Program Facts

The program analysis process infers abstract facts about the program. Consequently, to discuss the particular analyses introduced in this dissertation, this section introduces the representation of program facts. The representation is used to both show the current state of an analysis at a particular point in the program as well as to specify expectations of the analysis. Expectations serve as a specification for the behavior of the function that the analysis will attempt to verify.

The basic abstraction presented in this dissertation is the HOO abstraction, Heap with Open Objects. This abstracts a heap along with open objects. Consequently, facts are represented using the HOO notation. Figure 3.6 shows an abstract state represented with HOO, along with a corresponding program. The program in the figure creates a new object pointed to by `obj` that has two attributes: 'fld1' corresponds to the value 1 and 'fld2' corresponds to the value 2. The corresponding abstract state comes in two parts: the pre-condition (I) and the post-condition (II). Because all states represented by this variant of HOO are relative to an annotated precondition, HOO is a two-state abstraction; it abstracts a pair of states, one being the starting state, the other being the current state of the program. The relative precondition abstracts the starting state and is indicated in the lower right of the state (in this case (I)). The abstract state shown at (I) highlights the two components of the HOO domain. On the left there is a heap graph that shows that the program variable `obj` (in an dotted box in typewriter font) starts out pointing to a set of possible values, which is initially the singleton set containing undefined. On the right, in the dotted box, are constraints that place restrictions on sets that occur in the heap graph. Here there are no such restrictions.

(I) [`obj` → {undef} TRUE] (I)

```
obj = {};
obj["fld1"] = 1;
obj["fld2"] = 2;
```

(II) [`obj` → A | F : t → V $F = \{f_1\} \cup \{f_2\} \wedge f_1 =$ 'fld1' $\wedge f_2 =$ 'fld2' $\wedge V = \{v_1\} \cup \{v_2\} \wedge v_1 = 1 \wedge v_2 = 2 \wedge A = \{a\}$] (I)

Figure 3.6. – Demonstration of open object abstraction in a simple program

In the post-condition (II) the object has been created and its attributes initialized. Now an abstract object exists, shown as a table. The shaded top row is the set symbol for the base address of objects A. If this is not a singleton set, the object is a summary. In this case, I have constrained it to be a singleton set consisting of a single base address

a. Below the shaded top row are rows each describing a partition of attribute names for that object. Here I have decided to represent these two attribute names 'fld1' and 'fld2' using a single partition that conflates the two attribute names. This partition is represented with the set symbol F, where it is equated to the union of two singleton sets with attribute names f_i. Additionally, this partition has been assigned the attribute/value tracker t, which can keep track of specific attribute/value pairs from the beginning of the function to the end, as will be demonstrated in Section 3.4.1. Finally, the partition points to a set of values V that is made up of individual values v. Note that this is not the most precise abstraction because the two attributes have been summarized into a single partition. An alternative abstraction would construct a separate partition for each known attribute name [CCR14].

While each abstract state can, in fact, be viewed as the pair of the pre-condition and the current abstract state, for brevity, I show only the current abstract state along with the program point for the pre-condition. The actual pre-condition is provided at the indicated program point. It is important to note that symbols are shared between the pre-condition and current abstract state. Consequently, if a symbol is the same in both the pre-condition and the current abstract state, this means that the concrete values they represent are the same.

3.4. Example Analysis: JavaScript Class Implementation

This section demonstrates the analysis of the constructor of the `Class` example. To highlight the various challenges that the analysis encounters and addresses, I have broken down the analysis into three parts that are presented individually. For simplicity, each of these parts is presented as a whole analysis without assuming context that may actually be provided in `Class`. First, to demonstrate unknown attribute access, the read and write of objects in the `copy` function is presented. Next, the iteration over objects that occurs during the copy function is presented. Third, the call of the unknown client-supplied `init` function is analyzed. Finally, I present a summary of the whole constructor as could be analyzed with the techniques presented here.

3.4.1. Unknown Attribute Access

When the structure of an object is unknown, i.e. the attributes that the object may have, the number of attributes the object may have, and the corresponding values are unknown, it is difficult to determine what accessing attributes of these objects may do. To demonstrate how the HOO abstract domain handles this situation, consider the code shown in Figure 3.7. It reads from one object the value at attribute `p` and writes to another object also at attribute `p`.

For the purposes of demonstration, I assume a precondition with two separate objects at addresses a_1 and a_2. The structure of both of these object is completely unknown. The object at address a_1 has some attributes described by the set F_1 with corresponding values described by the set V_1 and an unknown mapping function t_1 from F_1 to V_1. The object at address a_2 is similar. The value of the attribute `p` is represented by the symbol

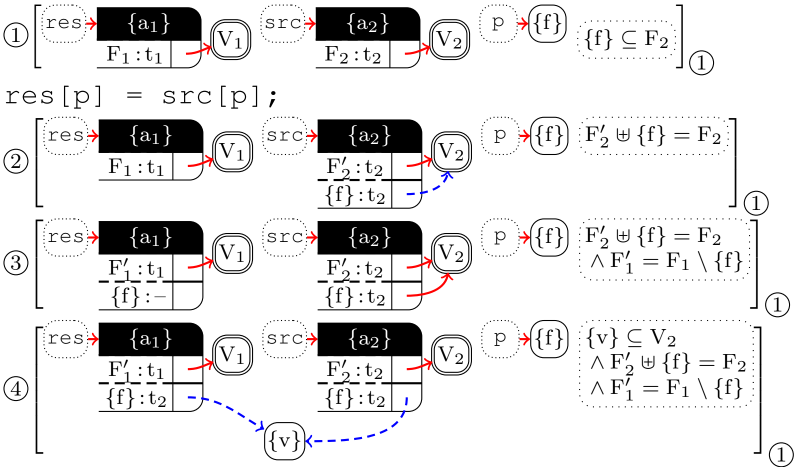


Figure 3.7. – Reading and writing from/to open objects with unknown attributes

f, but since f is unconstrained, it can take on any value and thus does nothing to dictate which attribute will be read or written.

It is critical when doing reads and writes of unknown objects that the analysis performs strong updates. By using strong updates, later if the same unknown attribute is accessed either by reading or writing, the same value can be retrieved. For example, if the value `res[p]` were read following this command, by performing strong updates the analyzer knows that the same value flowed from `src[p]` to `res[p]` and is now being read from `res[p]`.

The result of evaluating this compound command (both a read and a write) is shown in three steps, though in implementing the analysis the intermediate steps ② and ③ need not be actually produced. In ②, the read operation has been started by materializing the attribute that is being read. Because f is in F_2, the read must come from one of the attributes of F_2. Therefore, HOO extracts from F_2 the attribute f as a new partition of attributes in the object. This produces the F'_2, which is equal to F_2 without f along with the brand new partition that has exactly the same attribute/value tracker t_2 and points to the same set of values V_2.

Since this is also performing a write operation, the next step, shown in ③, begins the write operation by materializing exactly the attribute that is being written as its own partition in the object at address a_1. Unlike the read case, where the values and tracker are preserved, when writing to an object the specific value will soon be replaced with a new value and thus the exact value is unimportant. Because the object a_1 is open and new attributes can be added, it does not matter if the attribute already existed in the object before the write. Therefore, the new partition F'_1 for the existing attributes need not have contained f and thus f is removed from F_1 only if it existed already. This removal does not change the tracker t_1. The fact that the domain of the tracker is a superset of the set of possible attributes does not invalidate the tracker. However, the tracker is not transferred to the new partition.

The final step of this read/write operation, shown in ④, performs the actual read and write. Because there is now necessarily one partition that is being read and necessarily one partition that is being written, the value can now be transferred. Currently, however, there is only a summary of the values. Therefore, an individual value v must be materialized from V_2. Unlike the other materializations, because multiple attributes may have shared a single value, the value should not be removed from V_2, but simply a new constraint that $\{v\} \subseteq V_2$ is added. This minimizes case splits that may be required throughout the analysis. Once this value is split, clearly the read object points to this split off value because that is the value that was split off, and now the object at address A_1 now also points to this value. This completes both the read and the write operation.

There are several other cases that can occur when reading and writing, such as case splits caused by the presence of multiple partitions or reading of an unknown attribute. These more complex cases require careful handling and introduce a notion of complete and incomplete objects and will be covered in detail in Chapter 4.

In the end, HOO achieves a precise analysis of a read and write from one object to another even when the objects and attributes are completely unknown. It uses materialization to identify specific attributes and values that are being written, and it uses attribute/value trackers to precisely track which attributes map to which values. It thus performs relational abstraction and strong updates on unknown objects.

3.4.2. Object Attribute Iteration

The next key feature of the analysis is support for iterating over an object's attributes. In the example code, this occurs in the `copy` function which copies all of the attribute/value pairs from a source object pointed to by `src` to a result object pointed to by `res`, except for those attributes that are defined in an exclusion object pointed to by `exc`. The key challenge when analyzing code such as this is to be able to precisely infer a reasonable loop invariant that guarantees a precise postcondition when the analysis completes. This requires support for unknown attribute access along with support for iteration.

The main part of the `copy` function is shown in Figure 3.8. It contains the **`for-in`** loop that iterates over the attributes of the object pointed to by `src`. This figure shows key analysis states as would occur on the final iteration of abstract interpretation, when the already inferred invariants are simply being checked. The precondition for this analysis, which is normally either a developer-specified annotation, or provided by the context in a larger analysis is shown at program point ①. It describes three distinct objects at addresses a_1, a_2, and a_3. Each of these objects has a completely unknown structure, so this is a quite general precondition.

Appropriate partitioning of objects is vital for performing strong updates. To take advantage of strong updates across loop iterations, ② introduces two special attribute sets for iteration. The set F_i is the set of all attributes that have not yet been visited by the loop and is thus initially equal to F_2, whereas the set F_o is the set of all attributes that have already been visited by the loop and thus is initially empty. On each iteration an element is removed from F_i and placed into F_o, allowing relationships to represent not just the initial iteration of the loop, but also any iteration. We see these relationships in

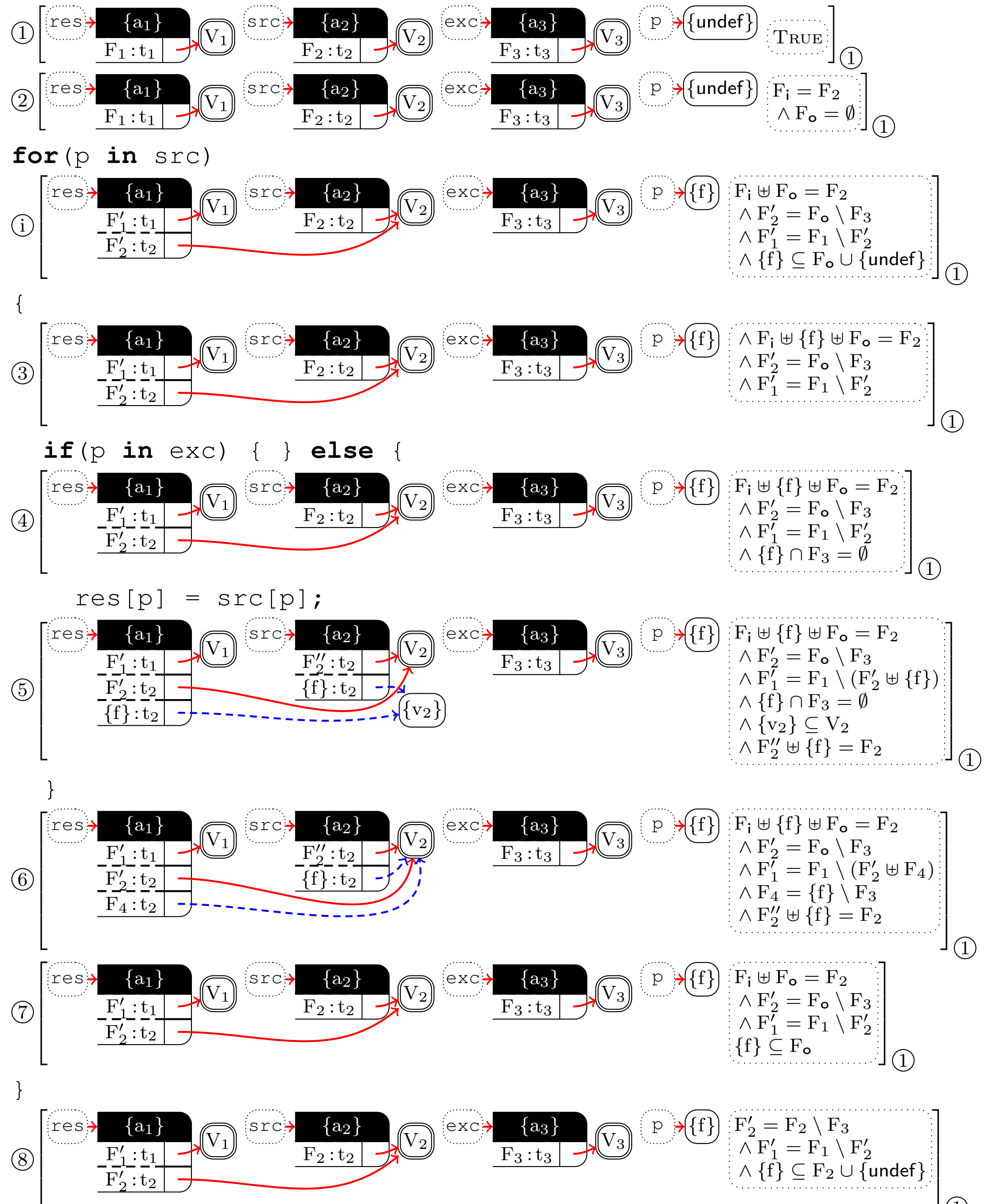


Figure 3.8. – Iterating over objects with unknown attributes

the loop invariant ⓘ.

The loop invariant ⓘ shows the two partitions of a_1 that are inferred by the analysis. The first partition F_1' is the initial set of attributes that were in a_1 minus those attributes added by the iteration process. The second partition F_2' is the set of attributes added by the iteration process. Both partitions are constrained by F_o, because the overwritten portion of a_1 can only be from the elements that have already been visited by the loop. Additionally, relationship between F_i and F_o can be more clearly seen in the loop invariant. They form a partition of the attributes of the object a_2. This loop invariant was inferred using a standard join/widening process in abstract interpretation [CC77].

Upon entering the body of the loop, as shown in ③, a new value f is bound to p. This value is selected from F_i and is thus disjoint from F_o. As a result, it is always the case that $F_i \uplus \{f\} \uplus F_o = F_2$. This ensures the invariant that f is contained in F_2 as well as ensuring that it is no longer included in either iteration set.

Depending on the value of f, one of two possible cases occurs. If the value of f is in F_3, it is in the exclusion object and nothing should happen. This ensures that anything in the exclusion set is not copied from the source to the result. In the other case, shown at ④, where f is disjoint from the exclusion object's attributes F_3, the attribute/value pair should be copied.

The copy itself proceeds like an unknown access as describe above. The key difference is that there are two partitions of the destination object. Thus, as shown in ⑤, a third partition for the result is added and f is removed from both existing partitions. However, because the partition F_2' necessarily cannot contain f, it is only actually removed from F_1', as reflected in the constraints on the side.

In ⑥, the two cases are joined. As a result, the materialized value v_2 is merged back into V_2. More important is the introduction of F_4, which is a conditional version of $\{f\}$. It is equal to $\{f\}$ if f is not in F_3. Otherwise, it is equal to the empty set. This precisely encodes the condition of the branch in the result of joining these two cases. The two different versions of the same objects are merged into one by pushing the distinguishing cases into the set constraints on the side.

The final step ⑦ of the loop body prepares for the end of the loop. It moves f from being a distinct element into F_o. This completes the transfer of a single unknown element from F_i to F_o. However, to do so, partitions had to be merged. The fact that F_4 was distinct from F_2' means that ⑥ can no longer be represented precisely without f, and since they have the same values, they can be merged into a new F_2'. Similarly, f is merged back into F_2'' to produce, once again F_2. This merging process summarizes the behavior of the loop into a form that is directly comparable with the loop invariant and thus the loop completes.

This process of materializing a single element for iteration from the not-visited set F_i, evaluating the loop body on that single element and then summarizing that element into the visited set F_o gives strong updates across loops. Essentially, each iteration establishes more of a relationship between objects where copies or other operations have occurred. Without separating out the visited set F_o, these relationships cannot exist because they do not exist at the beginning of the loop and they do not exist across the entirety of objects until the loop is complete.

3.4.3. Set Abstraction

The use of sets is central to the HOO abstraction. Every operation in some way manipulates sets. I have shown all of these manipulations in a dotted box at the side of HOO, but this reasoning is itself an abstraction. For HOO to function soundly, there are no requirements on what this abstraction is capable of. HOO is parametric with respect to the set abstraction. However, to realize the full benefits of HOO, the set abstraction has to be capable of a significant number of operations.

For example, in Figure 3.8, just to represent the loop invariant, the set domain must be capable of representing disjoint union, set difference, union, subset, and singleton sets of not just strings, such as attributes in JavaScript, but also special values such as `undef`. This is a significant number of operations. Additionally, the set abstraction needs to be able to infer invariants about sets and to compute joins over constraints of sets.

Because JavaScript does not have sets explicitly as values in the language and because the primary problem solved in this dissertation is not sets, I do not present the abstraction for sets here. There is a multitude of possible abstractions for sets that could be used with HOO and one that I use is presented in its entirety as a standalone analysis in Chapter 7. It supports not only all of the above operations, but is parametric with respect to values, so that any values can be placed within the sets and constraints on those values will be inferred.

3.4.4. Unknown Function Calls

Next, consider the analysis of the call to the user-supplied constructor. Here, I assume that the user-supplied constructor is provided in `init` and that the behavior of this constructor is unknown. In Figure 3.9, I show the abstract state immediately before, ⓐ, and immediately after, ⓑ, the callback to that constructor.

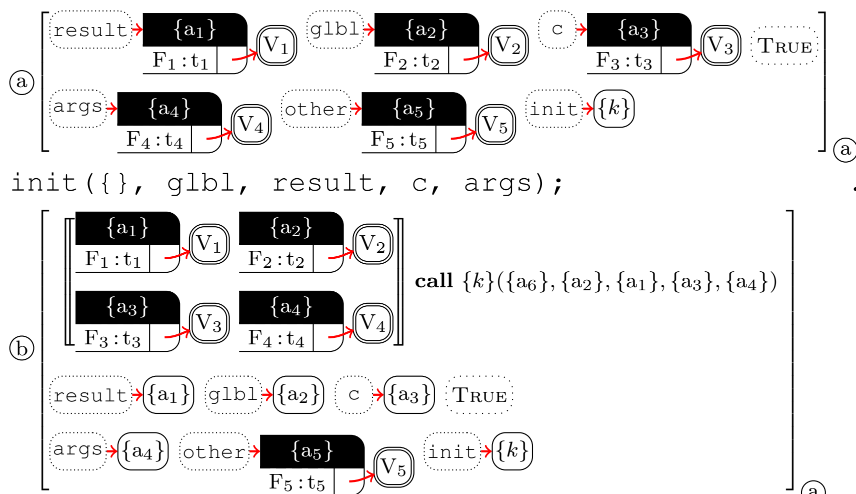


Figure 3.9. – Desynchronized terms are introduced by function calls to unresolvable functions

Immediately before the call to the constructor, there are five objects of primary concern: a_1 is the `result` object that is the class instance that is currently in the process of being constructed; a_2 is the global object that is threaded through all functions in programs; a_3 is the closure for this particular function; a_4 is the `args` object that is present for every function in JavaScript and stores the list of arguments passed to the function; and a_5 is another object that is local to the function and thus not accessible via a closure or the global object. This `other` object represents objects like `attrs`, which are protected objects that should not be externally accessible. Because there are many such objects in the program, I do no show all of them and instead show this `other` object as a placeholder that potentially represents many objects.

Analyzing the call is problematic. There is no annotation that dictates the behavior of the user-supplied constructor. To work around this, HOO splits the heap into two separate parts: (1) the part that may be reachable by the client-supplied constructor and (2) the part that is definitely not reachable by the client-supplied constructor. For the unreachable portion, there is no change and thus it is directly represented in the post-state ⓑ. For the reachable portion, the analysis handles the function call by desynchronizing the heap. This portion of the heap exists in a state prior to the function call.

The desynchronization process introduces a new element in the representation called a *desynchronized term* that is written $[\![H]\!]\,\textbf{call}\ \{k\}(\{a_6\},\{a_2\},\{a_1\},\{a_3\},\{a_4\})$, where H is the portion of the heap that is desynchronized and $\textbf{call}\ \{k\}(\{a_6\},\{a_2\},\{a_1\},\{a_3\},\{a_4\})$ is the function to be called and the arguments to be passed to resynchronize this portion of the heap with the surrounding portion. By introducing this desynchronized term, the post-state of the call can be written in such a way that, when the user-supplied constructor becomes known, such as when a function summary generated by HOO is reused, the now known function can convert the desynchronized heap back into a synchronized heap.

Once the heap has become desynchronized, the portion that is not contained in a desynchronized term is still accessible and usable. Consequently, it is possible to return $\{a_1\}$ even though the contents of the object at address $\{a_1\}$ are unknown (desynchronized). Portions that are contained in the desynchronized term are inaccessible until they are resynchronized by performing the corresponding function call. Desynchronized memory should not be accessed as it could lead to incorrect analysis results because that memory may have been updated, reallocated, deleted, or otherwise changed by the function call.

While JavaScript does not have many language protection mechanisms, developers have learned how to use local function variables effectively to provide significant protection of critical internal portions of code. The heap reachability query used when constructing a desynchronized term exploits the little language protection that is provided. Protected backups, such as the copying of `cfg` into `attrs` not only serve as protection from external mutation, but also ensure that objects like `other` are not included in the desynchronized term, which is important to ensure that subsequent calls to class instantiation behave the same (i.e. `attrs` after the call must match `attrs` before the call). Similarly, all local variables, such as `result` will not be included. However, five objects are desynchronized, due to their accessibility from the closure, the global object, or arguments as anything that is accessible may be mutated by the client-supplied constructor.

3.4.5. Bringing the Abstractions Together

The combination of all of the above analyses is an analysis for the constructor for `Class`. I elide the full preconditions and postconditions for the constructor due to their size. There are a significant number of closure objects that are used, created, and manipulated that complicate the representation.

As a result of support for unknown attribute access, attribute iteration, set abstraction, and unknown function calls, the abstractions presented in this dissertation are sufficient to infer precise postconditions and thus verify JavaScript library functions such as `Class`. The remainder of the dissertation details each of these abstractions with additional examples.

4. Heap Abstraction: Separation Logic with Open Objects

In this chapter I define the Heap with Open Objects (HOO) abstraction. It is an abstraction for a dynamic language heap combined with open objects. The presentation starts with necessary background on abstract interpretation sufficient to understand the purpose of HOO. Then, I introduce two versions of the HOO abstraction. First, the single-state abstraction, as presented in [CCR14], abstracts a single program state and thus is useful for verifying assertions in dynamic code. Second, the two-state abstraction abstracts two program states: a precondition and a current state and is thus suitable for inferring summaries of program behavior. After both abstractions are introduced, the various operations are defined for the two-state abstraction. The single-state HOO is sufficiently similar that it does not require a separate presentation.

Throughout this chapter I utilize the following symbols in the various definitions.

$$
\begin{aligned}
&\overline{\text{Address}} \subseteq \overline{\text{Value}}\\
&\overline{\text{Attribute}} \subseteq \overline{\text{Value}}\\
&v \in \overline{\text{Value}}\\
&f \in \overline{\text{Attribute}}\\
&d \subseteq \overline{\text{Attribute}}\\
&o \in \overline{\text{Object}} = \overline{\text{Attribute}} \rightharpoonup \overline{\text{Value}}\\
&\sigma \in \overline{\text{State}} = \overline{\text{Address}} \rightharpoonup \overline{\text{Object}}
\end{aligned}
$$

$\overline{\text{Address}}$ is the set of all concrete addresses, $\overline{\text{Attribute}}$ is the set of all concrete attributes (strings), and $\overline{\text{Value}}$ is the set of all values including addresses and attributes. $\overline{\text{Object}}$ is the set of partial functions from attributes to values, where unmapped attributes are not attributes in the object. Similarly concrete states are a partial function from addresses to objects. Individual concrete values v, attributes f, and object domains d are used in defining semantics. These partial functions are represented as association lists as shown in Chapter 3.

4.1. Abstract Interpretation Background

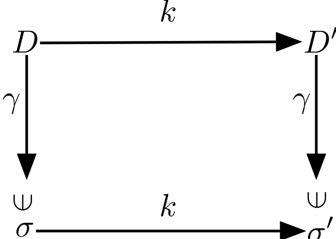
Abstract interpretation [CC76, CC77, CC79] is based upon a concrete interpreter of the language. A *concrete interpreter* takes program text and evaluates that program text starting from some state, which represents the computer's memory and input/output

systems. In doing so, it manipulates the state of the system. Each step in the interpretation transforms a concrete state σ of a system into an updated state σ' of the system via the $\langle\sigma\rangle\, k\, \langle\sigma'\rangle$ relation that was given in Section 3.1. Thus a concrete interpreter is a state transformer.

An *abstract interpreter* takes program text and evaluates that program text starting from some abstract state, which is a mathematical model of some artifact of the program (typically the set of reachable states). In doing so, it manipulates that mathematical model, producing a new mathematical model. These mathematical models D and D' respectively are elements of an *abstract domain* that defines the meaning of those models and provides operations on those models such as the abstract transfer functions $[D]\ k\ [D']$. Thus, an abstract interpreter is a transformer for elements of an abstract domain.

What abstract interpretation enables is the development of abstract domains that simultaneously reason about a large number of concrete states. Thus, the abstract interpretation can be used to determine the possible effects of a program without considering each concrete evaluation separately. When there are unboundedly or intractably many possible starting states to a program, the abstract domain can represent all of the states and through abstract interpretation can determine the behavior on any of the many inputs. Consequently, abstract interpretation gives a methodology for reasoning about all executions of a program simultaneously.

Of course, they key to doing this is abstraction. Abstraction means that it may not precisely represent any set of concrete states, but rather, an overapproximation of a set of concrete states. This overapproximation should be maintained throughout the abstract interpretation by ensuring that the transfer functions in the domain are sound with respect to the corresponding concrete transfer functions. Soundness is described by the following diagram:

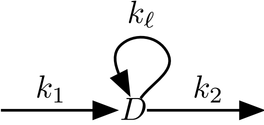


An abstraction D is related to a set of concrete states by the concretization function γ. The concretization function γ defines the meaning of an abstraction in terms of the concrete states it represents. This diagram shows that starting from an abstract domain element D and abstractly interpreting command k gives D'. If the same command k is concretely interpreted starting from any σ in the concretization of D, it produces a σ' that is in D'. This means that the set of all σ' are a subset of those described by $\gamma(D')$ and thus this is an overapproximation and is considered sound.

Of course, defining sound transfer functions $[D]\ k\ [D']$ is a challenge because of possibility of unbounded execution. When reasoning about any possible k, k can include loops. If the loop is a function of an unbounded input, the abstract interpretation, because

it considers any possible input must consider the loop repeating unboundedly many times. This, however, is an undesirable property because this means that performing an abstract interpretation may take an unbounded amount of time. As a form of analysis, it is desirable for the analysis to complete on any program regardless of its inclusion of loops.

The solution to this problem lies in the partial order that an abstract domain forms. The partial order is formed from ordering $\sqsubseteq$. This ordering must have the property that if $D_1 \sqsubseteq D_2$ then $\gamma(D_1) \subseteq \gamma(D_2)$. This means that the ordering of the abstract domain elements is related to the ordering of the subsets in the concrete through the γ function. With this partial order, it is possible to compute fixpoints [Tar55] for loops. A *fixpoint*, for the purposes of this discussion, is an element of the partial order (of the domain) such that, after interpreting the body of the loop, gives the same element of the partial order. In the following picture, a command k_1 happens preceding the loop. Then the loop body k_ℓ executes some number of times, followed by k_2, which happens after the loop:



The abstract domain element D is a fixpoint of this loop because k_1 produces an abstract domain element that is $\sqsubseteq D$, and starting from D, k_ℓ produces exactly the same domain element D. Therefore, the challenge in handling loops is computing fixpoints of the loops such as D.

In actuality, it is not necessary to compute a fixpoint of a loop. In fact, all that is required is a *post-fixpoint*, which has the property that if $[D]\ k_\ell\ [D']$, then $D' \sqsubseteq D$. Such a post-fixpoint D, if viewed logically, is an inductive *loop invariant* [Flo67, Hoa69]. Therefore, I use the term invariant generation to describe the process of computing these post-fixpoints.

The invariant generation process by abstract interpretation works like this: each command k is evaluated abstractly starting from a specified precondition (represented as an element of the abstract domain). If that command k is a loop, a loop invariant must be found. Initially, a candidate loop invariant D_0 is chosen to be the same as the abstract domain element immediately preceding the loop. Then the loop body k_ℓ is interpreted starting from D_0, giving D'_0. If the result of interpreting the loop body D'_0 is included in the candidate loop invariant ($D'_0 \sqsubseteq D$), the candidate invariant was inductive and is an actual invariant. Thus, the postcondition for the loop is the loop invariant. If not, the reachable abstract states that were reached by evaluating the loop body can be added to the candidate invariant via a join operation $\sqcup$, giving a new candidate invariant $D_1 = D_0 \sqcup D'_0$. This process can be repeated until a candidate $D'_i \sqsubseteq D_i$.

Unfortunately, while each join operation $\sqcup$ climbs the partial order, if the join operation is actually the least upper bound of the two inputs and the partial order has infinite height (as many do, such as intervals), this process may not terminate. There can be an unbounded number of iterations required to reach a fixpoint. To get around this, abstract interpretation requires a widening operator $\triangledown$, which works like the join operator but climbs the partial order in potentially big leaps ensuring that only a finite number of

iterations is required to reach a post fixpoint. In doing so, the guarantee that the most precise inductive invariant will be found is lost, but with appropriately designed operators, many found invariants will be acceptably precise — precise enough that they can prove interesting properties of the program.

As a result of the above process, an abstract domain is defined to be a partial order that has the following key operations defined for it:

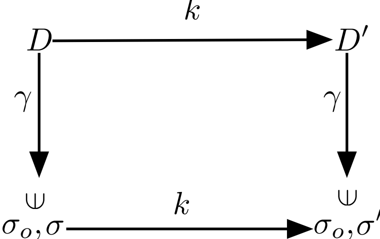
- Concretization: The function γ maps an abstract domain element to a set of concrete states.
- Transfer functions: The functions $[D]\ k\ [D']$ that soundly derive a domain element D' from a domain element D by abstractly interpreting the command k.
- Abstract inclusion: The $D_1 \sqsubseteq D_2$ operation that determines if D_1 is less than D_2 in the partial order defined by the abstraction partial order. Intuitively D_1 is less than D_2 if it is more precise.
- Join: The $D_1 \sqcup D_2$ operation that yields a domain that is an overapproximation of the least upper bound of D_1 and D_2.
- Widening: The $D_1 \nabla D_2$ operation that a domain that is an overapproximation of the least upper bound of D_1 and D_2 and is guaranteed to converge (reach a post fixpoint) within a finite number of loop iterations.

The definition of the HOO abstract domain contains all of these necessary elements starting in the next section.

Non-standard abstractions Abstractions need not abstract a set of concrete program states. They can, in fact, abstract any artifact of program execution. This means that anything that can be described by a collecting semantics [Shi91] can be abstracted. Examples include complexity analysis [GMC09], as well as abstracting program traces [RM07], and ranking functions [Urb13].

In this thesis I am concerned with one particular variety of non-standard abstraction. The abstractions used by the HOO come in two forms: The first is a single-state abstraction that is standard according to the previously given definition. It abstracts a set of concrete program states. The second is a non-standard, two-state abstraction. Rather than abstracting a standard concrete semantics, it abstracts a special semantics that keeps track of two states. The first state is a pre-state for the function being analyzed. This state is never updated. It is simply carried through the program execution. The second state is a current state that may be reached from that particular pre-state. Such an abstraction works as in the following diagram.

As before, the abstract domain element D transitions to D' when k is interpreted. The key difference is that γ gives a pair of concrete states σ_o, σ. The first state σ_o represents an initial state whereas σ represents the current state in evaluation. By interpreting k starting from σ, the interpreter gives σ', which when paired with the same σ_o, must be in $\gamma(D')$.



4.2. Representation: Heap with Open Objects

In this section, I give the representation and concretization of the two variants of HOO. This defines the meaning of each HOO abstraction and gives the idea of what each is capable of representing. However, because all of the domain operations are essentially the same for both the single-state and the two-state versions of HOO, I do not give two definitions of all of the operations. In subsequent sections the operations are given in terms of the two-state version of HOO, but they can be trivially adapted to the single-state version by ignoring all of the first state operations, which are minimal.

4.2.1. Single-State Heap with Open Objects

Single-state HOO is an abstraction for a dynamic language heap, which represents the state of a dynamic language program. HOO has two components: a heap graph represented as separation logic, and a pure constraint that restricts the values of symbols that appear in the heap graph. What makes HOO unique is its pure component is second-order. It restricts symbols that represent sets of values. These symbols can be any of the following:

$$\{\mathrm{a}\}, \{\mathrm{f}\}, \{\mathrm{v}\}, \mathrm{A}, \mathrm{F}, \mathrm{V} \in \overline{\mathrm{Symbol}}$$

where A represents a set of addresses, F represents a set of attributes, and V represents a set of values. The $\{\mathrm{a}\}$, $\{\mathrm{f}\}$, and $\{\mathrm{v}\}$ sets are the respective singleton forms.

Definition 1 (Heap with Open Objects). *A heap with open objects is an abstraction represented with the following logical syntax:*

$$\begin{aligned} \overline{\mathrm{Heap}} \ni H &::= H_1 * H_2 \mid \mathrm{A} \cdot \langle O \rangle \mid \textsc{Emp} \\ \overline{\mathrm{Object}} \ni O &::= O_1 ; O_2 \mid \mathrm{F} \mapsto \mathrm{V} \mid \textsc{None} \\ \overline{\mathrm{Domain}} \ni D &::= D_1 \vee D_2 \mid H \between P \end{aligned}$$

An abstract domain element D is either a disjunction of abstract domain elements, or a heap H restricted by a pure domain element for sets P. This element is of a domain that is a parameter to the abstraction. An individual heap H is a standard separation logic heap consisting of two disjoint parts combined with separating conjunction, a set of objects $\mathrm{A} \cdot \langle O \rangle$ *at addresses described by* A *with structure O, or the empty* Emp*. Objects are also structured with a separating conjunction represented with* ; *where attribute sets are known to be separate from all other attribute sets. Alternatively, objects consist of a map from a set of attributes* F *to a set of values* V *or they are the empty object.*

The resulting abstraction is a reduced product [CC79] between a heap abstract domain element H and a set abstract domain element P. The set domain is used to represent relationships between sets of attributes of objects. The information from the set domain affects points-to facts $\mathrm{A} \cdot \langle \mathrm{F} : \mathrm{t} \mapsto \mathrm{V} \rangle$ by constraining the sets of addresses A, attributes F, and values V. Therefore the meaning of a HOO abstract state is closely tied to the meaning of set constraints. Since HOO is parametric with respect to the abstract domain for sets (an example domain is given in Chapter 7), its concretization is given in terms of a concretization for the set domain γ_P:

$$\begin{aligned}\gamma_P : \overline{\mathrm{SetDomain}} &\rightarrow \wp(\overline{\mathrm{Valuation}}) \\ \text{where } \overline{\mathrm{Valuation}} &= \overline{\mathrm{Symbol}} \rightharpoonup \wp(\overline{\mathrm{Value}})\end{aligned}$$

A valuation $\eta \in \overline{\mathrm{Valuation}}$ is a partial function that maps each symbol used in the heap to a set of concrete values. These values are constrained by the abstract domain for sets, consequently, any element of the set domain P concretizes to a set of possible valuations.

$$\begin{aligned}
&\gamma : \overline{\mathrm{Object}} \rightarrow \wp(\overline{\mathrm{Valuation}} \times \overline{\mathrm{State}} \times \wp(\overline{\mathrm{Attribute}})) \\
&\gamma(O_1; O_2) = \left\{ \eta, o, d \;\middle|\; \begin{array}{l} \exists o_1, o_2, d_1, d_2 . (\eta, o_1, d_1) \in \gamma(O_1) \\ \wedge\, (\eta, o_2, d_2) \in \gamma(O_2) \wedge o = o_1 \uplus o_2 \\ \wedge\, d = d_1 \uplus d_2 \end{array} \right\} \\
&\gamma(\mathrm{F} \mapsto \mathrm{V}) = \left\{ \eta, o, d \;\middle|\; d = \eta(\mathrm{F}) \wedge \exists f.\ f \in \eta(\mathrm{F}) \wedge o(f) \in \eta(\mathrm{V}) \right\} \\
&\gamma(\textsc{None}) = \{ \eta, [], \emptyset \}
\end{aligned}$$

$$\begin{aligned}
&\gamma : \overline{\mathrm{Heap}} \rightarrow \wp(\overline{\mathrm{Valuation}} \times \overline{\mathrm{State}}) \\
&\gamma(H_1 * H_2) = \left\{ \eta, \sigma \;\middle|\; \begin{array}{l} \exists \sigma_1, \sigma_2 .\ (\eta, \sigma_1) \in \gamma(H_1) \wedge (\eta, \sigma_2) \in \gamma(H_2) \\ \quad \wedge\, \sigma = \sigma_1 \uplus \sigma_2 \end{array} \right\} \\
&\gamma(\mathrm{A} \cdot \langle O \rangle) = \left\{ \eta, \sigma \;\middle|\; \begin{array}{l} \forall a \in \eta(\mathrm{A}).\ \exists o, d . \sigma(a) = o \\ \quad \wedge\, (\eta, o, d) \in \gamma(O) \wedge \mathrm{Dom}\,(o) = d \end{array} \right\} \\
&\gamma(\textsc{Emp}) = \{ \eta, [] \}
\end{aligned}$$

$$\begin{aligned}
&\gamma : \overline{\mathrm{Domain}} \rightarrow \wp(\overline{\mathrm{Valuation}} \times \overline{\mathrm{State}} \times \overline{\mathrm{State}}) \\
&\gamma(D_1 \vee D_2) = \left\{ \eta, \sigma \;\middle|\; (\eta, \sigma) \in \gamma(D_1) \vee (\eta, \sigma) \in \gamma(D_2) \right\} \\
&\gamma(H \mathbin{\dot{|}} P) = \left\{ \eta, \sigma \;\middle|\; (\eta, \sigma) \in \gamma(H) \wedge \eta \in \gamma_P(P) \right\}
\end{aligned}$$

Figure 4.1. – Concretization of the HOO abstract domain

Based on this concretization for sets, Figure 4.1 shows the concretization of a single-state heap based on HOO. As a result of its dependence on sets, the valuation η maps symbols to sets of values, which are, in effect, summaries. Multiple objects can be summarized by

representing their addresses in the same set. Multiple attributes can be summarized by representing these attributes in the same set and using that set to describe a partition of attributes in the object. Because these sets are modeled as symbols and are not simply a result of expressing a collecting semantics of the analysis, a separate pure domain for sets can relate set symbols to one another, constraining objects in the heap. Consequently, HOO can abstract dynamic language heaps directly through separation logic without requiring the object attribute names to be fixed.

Example 4.1 (HOO Parameterization). HOO is a parametric heap abstraction that takes a set abstraction as a parameter and uses that to represent the various addresses, attributes, and values in the heap. In Figure 4.2, HOO is shown instantiated with three

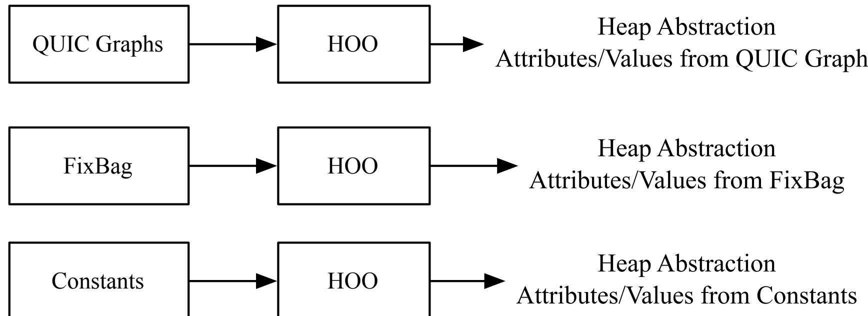


Figure 4.2. – HOO is a parametric heap abstraction that can be instantiated with any abstraction for sets of addresses, attributes and values.

different domains for sets. On top, HOO is instantiated with a QUIC-graphs-based domain as described in Chapter 7. This produces a heap graph with all addresses, attributes, and values represented as sets. In the middle, FixBag [PTTC11] is a different representation of sets with a completely different internal representation, but as long as its interface conforms, it can be used with HOO. Similarly, other domains like, constants domains can be used along with HOO. In the end, each of these heaps has different precision and different performance because HOO relies upon the underlying abstraction to determine the representation it uses for a given set of heaps.

The single-state version of HOO is useful for proving properties about object structures. Specifically it is ideal for representing relationships between multiple objects in the heap. If, for example, two objects in the heap have the same attributes and the same values, HOO can represent the equality of the attribute sets of those two objects and the value sets of those two objects. When trying to prove that objects have certain attributes or values (as might be checked by an assertion), this type of abstraction is ideal. However, if the goal is to infer summaries of functions or library code, the single-state abstraction is insufficient.

4.2.2. Two-State Heap with Open Objects

To be able to infer function summaries, as would be useful for determining the behavior of a function through analysis, two states must be abstracted. In the two-state abstraction, one state is a key program point and the other state represents the current program point. For instance, the first state could be the pre-condition to a library call and the second state could be the state of the program relative to that pre-condition at its program point. In the concrete, a two-state abstraction approximates a pair of states, where the second state may be reachable from the first from an associated program point. While possible to extend abstractions to full traces [Riv05], the two-state abstraction is sufficient for the problems tackled in this dissertation.

Definition 2 (Two-State HOO). *Two-State HOO uses the same heap and object syntax as HOO, but has a new definition of domain:*

$$\overline{\text{Domain}} \ni D ::= D_1 \vee D_2 \mid [H_2]_{H_1} \mathrel{\vdots} P$$

Instead of using a single heap H, there are now two heaps H_1 and H_2. The first state is H_1 and indicates the state at the pre-condition of the library function being analyzed. The second state H_2 represents the state of the program at the current program point. These states can be related to one another through the pure domain P.

In the two-state HOO, the parametric single pure domain element P is critical for representing relationships. If the same symbol occurs in both states, because it is restricted by a single P, it must be bound to the same value. Similarly P allows two different symbols appearing in two different states to be related through set constraints.

Example 4.2 (Two-state abstraction). In the following state, there are two abstract heaps and a single pure domain element.

$$\left[\{\mathrm{a}\} \cdot \langle \mathrm{F}' \mapsto \{\mathrm{v}\}\rangle\right]_{\{\mathrm{a}\}\cdot\langle \mathrm{F}\mapsto\{\mathrm{v}\}\rangle} \mathrel{\vdots} \mathrm{F}' \subseteq \mathrm{F}$$

This represents two concrete states. This constrains the relationship between those states so that they both refer to the same object because they use the same symbol $\{\mathrm{a}\}$ and the number of attributes has been possibly reduced: an attribute may have been deleted. All other attributes remain the same and no other attributes can have been observably added (added and then later removed is acceptable).

In addition to extending HOO to abstract multiple states, I also extend HOO with a new element called an attribute/value tracker. An attribute/value tracker relates specific attributes to specific values within a partition of an object. An attribute/value tracker is a name that is associated with a partial function from the attributes defined on a given partition to values that are contained in the set of associated values. By associating a tracker name with multiple partitions in multiple objects in two states, allows particular relationships to be tracked from the beginning of a function to a later point in a function, including the end.

Definition 3 (Attribute/value trackers). *Attribute/value trackers modify objects in HOO adding an optional tracker* t *to each mapping from sets of attributes* F *to sets of values* V*:*

$$\overline{\text{Object}} \ni O ::= O_1; O_2 \mid \mathrm{F} : \mathrm{t} \mapsto \mathrm{V} \mid \mathrm{F} : - \mapsto \mathrm{V} \mid \textsc{None}$$

The tracker is an uninterpreted function that maps each element from the corresponding F *to an element represented by the corresponding* V*. An elided tracker – indicates an unidentified relationship.*

The concretization of HOO with attribute/value trackers, due to both the multi-state abstraction and the presence of attribute/value trackers is different from basic HOO. First, a domain element D concretizes to a set of triples, including two concrete states σ_1, σ_2 and valuation η. Because the pure domain constrains relationships between the two abstract states, a common valuation is shared between the two concrete states. Second, the symbol μ for the binding of the trackers is constrained throughout the concretization:

$$\begin{aligned} &\mathrm{t} \in \overline{\text{TrackSym}} \\ &\mu \in \overline{\text{TrackerMap}} = \overline{\text{TrackSym}} \rightharpoonup \overline{\text{Attribute}} \rightharpoonup \overline{\text{Value}} \end{aligned}$$

An element $\mu \in \overline{\text{TrackerMap}}$ maps a tracker symbol to a partial function from attributes to values. The domain of that function is fixed when the tracker is introduced.

The definition of concretization is given in Figure 4.3. In addition to the standard separation logic operations as defined by HOO, the tracker map μ is threaded throughout the concretization, ensuring that it is shared between all partitions of attributes of all objects for both heaps that are described by the two-state abstraction.

Points-to Shorthand Notation In JavaScript and other dynamic languages, there are two classes of pointers: pure pointers and object pointers. Pure pointers are used to represent stack-allocated local variables. Stack-allocated local variables point to either objects or directly to values and as such they are simple pointers. Object pointers are the more complex form where an object contains attributes and each attribute points to a corresponding value.

To simplify the abstraction and to unify these two classes of pointers, HOO uses a special unit attribute that is written (). This unit attribute is used to represent a pure pointer as an object pointer. Thus the only kind of pointer that need be represented by HOO is an object pointer. However, because it is cumbersome and unhelpful to write this unit attribute everywhere, I write pure pointers that can be translated into the equivalent object pointer using the unit value. For example, consider the following bit of heap (in either single- or two-state HOO).

$$\{\mathrm{a}_1\} \mapsto \{\mathrm{v}_1\} * \{\mathrm{a}_2\} \mapsto \{\mathrm{v}_2\}$$

These two pure pointers can be translated into the equivalent object pointer representation where each source is actually an object with a single attribute () and each destination is

$$
\begin{array}{l}
\gamma : \overline{\text{Object}} \rightarrow \wp(\overline{\text{Valuation}} \times \overline{\text{TrackerMap}} \times \overline{\text{State}} \times \wp(\overline{\text{Attribute}})) \\
\gamma(O_1; O_2) \quad = \left\{ \eta, \mu, o, d \;\middle|\; \begin{array}{l} \exists o_1, o_2, d_1, d_2. \\ \quad (\eta, \mu, o_1, d_1) \in \gamma(O_1) \\ \quad \wedge (\eta, \mu, o_2, d_2) \in \gamma(O_2) \\ \quad \wedge o = o_1 \uplus o_2 \wedge d = d_1 \uplus d_2 \end{array} \right\} \\
\gamma(\mathrm{F} : \mathrm{t} \mapsto \mathrm{V}) = \left\{ \eta, \mu, o, d \;\middle|\; \begin{array}{l} d = \eta(\mathrm{F}) \wedge \forall f \in \eta(\mathrm{F}). \\ \quad o(f) \in \eta(\mathrm{V}) \wedge \mu(\mathrm{t})(f) = o(f) \end{array} \right\} \\
\gamma(\mathrm{F} : - \mapsto \mathrm{V}) = \left\{ \eta, \mu, o, d \;\middle|\; d = \eta(\mathrm{F}) \wedge \forall f \in \eta(\mathrm{F}).\ o(f) \in \eta(\mathrm{V}) \right\} \\
\gamma(\textsc{None}) \quad = \{ \eta, \mu, [], \emptyset \}
\end{array}
$$

$$
\begin{array}{l}
\gamma : \overline{\text{Heap}} \rightarrow \wp(\overline{\text{Valuation}} \times \overline{\text{TrackerMap}} \times \overline{\text{State}}) \\
\gamma(H_1 * H_2) = \left\{ \eta, \mu, \sigma \;\middle|\; \begin{array}{l} \exists \sigma_1, \sigma_2.\ (\eta, \mu, \sigma_1) \in \gamma(H_1) \\ \quad \wedge (\eta, \mu, \sigma_2) \in \gamma(H_2) \wedge \sigma = \sigma_1 \uplus \sigma_2 \end{array} \right\} \\
\gamma(\mathrm{A} \cdot \langle O \rangle) \quad = \left\{ \eta, \mu, \sigma \;\middle|\; \begin{array}{l} \forall a \in \eta(\mathrm{A}).\ \exists o, d. \\ \quad \sigma(a) = o \wedge (\eta, \mu, o, d) \in \gamma(O) \wedge \mathrm{Dom}\,(o) = d \end{array} \right\} \\
\gamma(\textsc{Emp}) \quad = \{ \eta, \mu, [] \}
\end{array}
$$

$$
\begin{array}{l}
\gamma : \overline{\text{Domain}} \rightarrow \wp(\overline{\text{Valuation}} \times \overline{\text{State}} \times \overline{\text{State}}) \\
\gamma(D_1 \vee D_2) \;= \left\{ \eta, \sigma_1, \sigma_2 \;\middle|\; (\eta, \sigma_1, \sigma_2) \in \gamma(D_1) \vee (\eta, \sigma_1, \sigma_2) \in \gamma(D_2) \right\} \\
\gamma([H_2]_{H_1} \mathrel{\vdots} P) = \left\{ \eta, \sigma_1, \sigma_2 \;\middle|\; \begin{array}{l} \exists \mu.\ (\eta, \mu, \sigma_1) \in \gamma(H_1) \\ \quad \wedge (\eta, \mu, \sigma_2) \in \gamma(H_2) \wedge \eta \in \gamma_P(P) \end{array} \right\}
\end{array}
$$

Figure 4.3. – Concretization of two-state HOO abstractions with attribute/value trackers

the corresponding value pointed to by that attribute. Trackers are unnecessary as the only attribute is the () attribute. The resulting, equivalent form is the following:

$$\{a_1\} \cdot \langle\{()\} : - \mapsto \{v_1\}\rangle \ast \{a_2\} \cdot \langle\{()\} : - \mapsto \{v_2\}\rangle$$

In this way formulas are significantly simplified, while keeping the abstraction simpler to define domain operations.

Graphical Notation For expository purposes, I most often use a graphical representation of abstract states. This notation was used to introduce the problems solved in this dissertation in Chapter 3. In this graphical notation, there are four kinds of boxes. (1) A dotted box with a `typewriter` font symbol is a symbol representing a stack-allocated memory address and is thus a representation of a local variable. Arrows out of these boxes represent pure points-to shorthand as described above. (2) A double-lined box with an A, F, or V in the box represents a summary of values. If those values are addresses (as in A), those addresses will not be dereferenced, as the target object is undefined. Consequently, these boxes do not have arrows initiating at the box. (3) A table represents an object and its partitions along with all of the mappings to values. The top row of a table, with a black background, indicates the symbol that is the address of the object. In each row, there are two columns. The first column contains the symbol for the attribute set and the tracker, if defined. The second column contains the value symbol, or an arrow to the box that contains the value symbol. (4) A dotted box with a logical constraint inside is the pure constraints that restrict the symbols used for addresses, attributes, and values using the set abstraction.

The graphical notation can be translated into the logical notation by transforming objects into their logical equivalent, where multiple objects and points-to relationships are joined via separating conjunction. The previous abstract state shown as the right-hand-side subscript only represents the heap and not the pure part. The pure part is captured entirely within the second abstract state shown in brackets. The following shows how points-to arrows are translated from the graphical notation into the corresponding logical representation:

Graphical	$\Longrightarrow$	**Logical**
`r` → {a}	$\Longrightarrow$	$\mathtt{r} \mapsto \{a\}$
A: $F_1 : t_1$ → V_1; $F_2 : t_2$ → V_2	$\Longrightarrow$	$A \cdot \langle F_1 : t_1 \mapsto V_1; F_2 : t_2 \mapsto V_2\rangle$

Example 4.3 (Graphical Notation). In Figure 3.6, the following abstract state was shown graphically:

$$\left[\mathtt{obj} \to A\langle F:t \to V\rangle \quad \begin{array}{l} F = \{f_1\} \cup \{f_2\} \land f_1 = \text{'fld1'} \\ \land f_2 = \text{'fld2'} \land V = \{v_1\} \cup \{v_2\} \\ \land v_1 = 1 \land v_2 = 2 \land A = \{a\} \end{array} \right]_{\mathtt{obj} \to \{\text{undef}\}}$$

Previously, the pre-state was shown implicitly with a number, whereas here it is shown explicitly. This is equivalent to the following separation logic formula that is represented by the two-state HOO abstract domain:

$$
\begin{aligned}
&[\texttt{obj} \mapsto \mathrm{A} * \mathrm{A} \cdot \langle \mathrm{F} : \mathrm{t} \mapsto \mathrm{V} \rangle]_{\texttt{obj} \mapsto \mathrm{V}_u} \\
&\,\mathrm{F} = \{\mathrm{f}_1\} \cup \{\mathrm{f}_2\} \wedge \mathrm{f}_1 = \text{'fld1'} \\
&\wedge \mathrm{f}_2 = \text{'fld2'} \wedge \mathrm{V} = \{\mathrm{v}_1\} \cup \{\mathrm{v}_2\} \\
&\wedge \mathrm{v}_1 = 1 \wedge \mathrm{v}_2 = 2 \wedge \mathrm{A} = \{\mathrm{a}\} \\
&\wedge \mathrm{V}_u = \{\mathsf{undef}\}
\end{aligned}
$$

4.3. Materialization with Set Abstraction

The abstract transfer functions are defined to only operate on singleton objects and attributes. Therefore, before defining the transfer functions, it is necessary to define materialization. *Materialization* is responsible for separating a single element out of a summary of possibly many elements. This is critical because, when JavaScript manipulates an object, it does not manipulate a summary of objects, but rather one object at a time. Similarly it manipulates one value at a time or one attribute at a time. However, in HOO, which may lump many objects, attributes, and values together into a single set symbol, that set symbol must be split into the necessary singleton symbol to be transformed by the program and the remaining summary that is left after removing the singleton symbol.

Without materialization, it is necessary to perform weak updates [Deu94] instead of strong updates. A strong update replaces an old abstract value with a new one by first removing the old value and then adding the new one. In contrast, a weak update simply adds a new abstract value as a possibility without first removing the old. It is well known that weak updates can cause significant precision loss during analysis [SRW02]. Fortunately, because HOO supports materialization, it is possible to perform strong updates.

The following two examples demonstrate the combination of materialization with transfer functions for storing to an object and loading from an object. They are intended to convey the intuition of materialization before it is formally defined.

Example 4.4 (Attribute Materialization for Store). Attribute materialization for store operations is simple. Since the value of the particular attribute is about to be overwritten, there is no need to preserve the original value. The implementation of store is shown in Figure 4.4.

Store looks up the corresponding objects to x_1, x_2, and x_3 in ⓐ, which in this case are $\{\mathrm{a}_1\}$, $\{\mathrm{f}\}$, and $\{\mathrm{v}\}$ respectively. Attribute materialization then iterates through each partition in $\{\mathrm{a}_1\}$ and reconstructs the partition by removing $\{\mathrm{f}\}$ from the partition. If $\{\mathrm{f}\}$ was not already present in the partition, this represents no change; otherwise it explicitly removes $\{\mathrm{f}\}$. Finally, after all of the existing partitions have been reconstructed, a new partition for $\{\mathrm{f}\}$ is created and it is pointed to the stored value $\{\mathrm{v}\}$ giving ⓑ. By

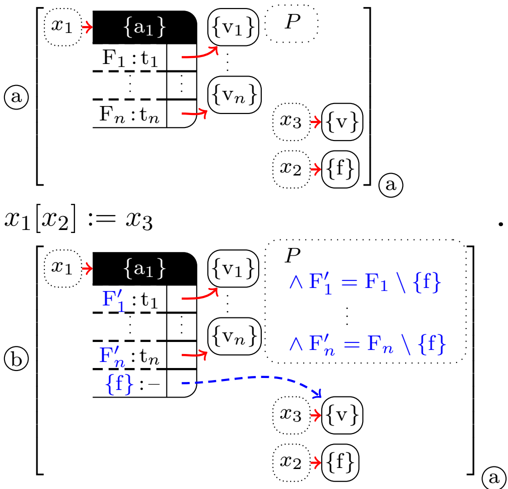


Figure 4.4. – Attribute materialization for storing to objects

performing this attribute materialization, subsequent reads of the same property {f}, even if its concrete value is unknown, are guaranteed to be directed to {f} and thus store performs strong updates.

Example 4.5 (Attribute Materialization for Load). Attribute materialization for load is similar to store. The key difference is that there is a possible result for each partition of the read object. The HOO abstract domain uses a finite disjunction to represent the result of this case split as shown in Figure 4.5.

A load operation must determine which, if any, of the partitions the attribute {f} is in. In the worst case, it could be in any of the partitions and therefore a result must be considered for each case. In each case, {f} is constrained to be in that particular partition and therefore in no other partition. If this is inconsistent under the current analysis state, the abstract state will become bottom for that case and it can be dropped. The default case, which implicitly represents all attributes not currently in the object, must be considered, as a possible source for materialization if there is a chance the attribute does not already exist in the object. Such a materialization does not explicitly cause any repartitioning.

Note that when materialization splits a partition, it copies the attribute/value tracker into each of the new partitions. This ensures that, if the set abstraction is precise, the materialization operation is completely precise.

Materialization is typically applied before a transfer function so that the precondition of the transfer function is met. The materialization operation attempts to split off a specific singleton symbol from a summary symbol. If it succeeds, the abstraction has that singleton symbol explicitly represented in the heap abstraction. If it fails, the singleton symbol does not exist and summary symbols are constrained to definitely not contain the singleton symbol of interest. These two cases are represented throughout the materialization rules shown in Figure 4.6. All of the rules support multiple results in

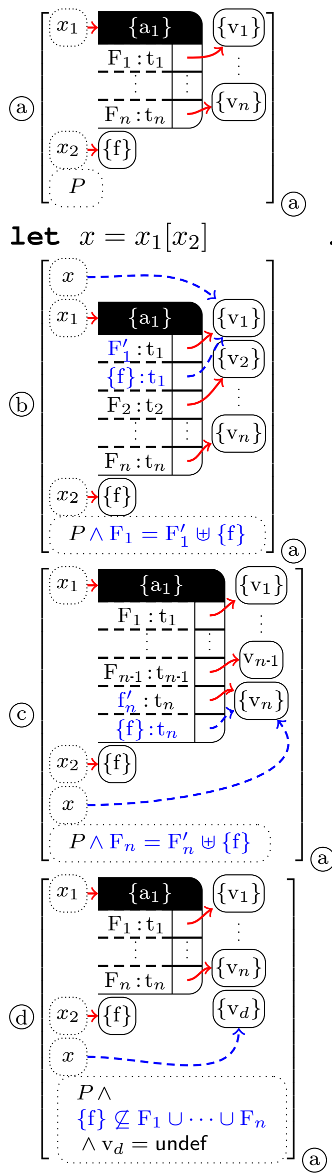


Figure 4.5. – Attribute materialization for loading from objects

order to allow case splits that determine if a singleton symbol is definitely in or definitely not in a summary symbol.

Materialization rules are of the form $D_1 \Rightarrow D_2$ and thus intended to be used with the rule of consequence from Hoare logic [Hoa69] to allow a future rule to be applied. For example, rules for assignment (next section) can only be applied to singleton object addresses, singleton attributes, and often singleton values. By applying materialization correctly, an abstract heap element that consists of summary object addresses, summary attributes, and summary values can be converted to the appropriate singleton form without loss of precision, assuming a precise pure domain.

There are three different judgments forms that are responsible for materialization. The first is responsible for materializing within a single object O given a pure set domain P. These rules produce a set of possible object/pure set domain pairs that are utilized by the second judgment form. The second judgment form is responsible for materialization that happens within a heap H given a pure set domain P. The result of these judgments is a set of heap and pure set domain pairs that represent the possible ways a particular element could be materialized. This set of pairs is utilized by the third judgment form. The third judgment form performs materialization on an abstract domain element D and yields another abstract domain element. Because abstract domain elements can represent disjunctions, sets of results are not required in transfer functions.

There are three rules that follow the first judgment form. The MAT-VALUE rule materializes a value from a summary. It is applied to a single attribute partition that has a singleton attribute $\{\mathrm{f}\}$ and a tracker t that points to a summary value V. Any value that is contained within the summary V can be materialized and as such some fresh v is introduced that is the sole value that corresponds to this singleton attribute. The MAT-ATTR rule materializes a specific attribute from an attribute summary. Note that this rule does not require values to be non-summary elements, so it can be applied to an attribute partition regardless of the corresponding value or attribute/value tracker. This rule captures two cases. Either the required attribute f is in this particular attribute partition F or it is not. In the first case, two new partitions are produced that take the place of the old partition. A new attribute F' is introduced that represents the residual after removing f from the F partition. Note that attribute/value trackers are cloned in this case as both new partitions represent a subset of the previous partition F, and thus the same tracker can be used. The second case assumes that f is not from F and no new partitioning is produced. The final rule MAT-OBJ-FRAME is the intra-object frame rule. Just like $*$ works for addresses, the ; operator works within objects. Therefore a frame rule applies. Using the frame rule, materialization can be performed on an isolated part of an object. All other parts of the same object remain untouched and unaffected by the materialization.

The second judgment form also has three rules. The MAT-OBJ rule is responsible for materialization within an object. It makes use of the first judgment form to materialize within an object and it simply introduces the corresponding case splits at the heap level by making the object at the singleton address a be an object in each of the results of object materialization. The MAT-ADDR rule is responsible for materializing a specific address from a summary address. It works much like MAT-ATTR producing two cases:

$$\boxed{O ¦ P \Rightarrow \wp(O ¦ P)}$$

Mat-Value

$$\frac{\text{v is fresh} \qquad P' = P \wedge \{\text{v}\} \subseteq \text{V}}{\{\text{f}\} : \text{t} \mapsto \text{V} ¦ P \Rightarrow \left\{\ \{\text{f}\} : \text{t} \mapsto \{\text{v}\} ¦ P'\ \right\}}$$

Mat-Attr

$$\frac{\text{F}' \text{ is fresh} \qquad P' = P \wedge \{\text{f}\} \uplus \text{F}' = \text{F} \qquad P'' = P \wedge \{\text{f}\} \cap \text{F} = \emptyset}{\text{F} : \text{t} \mapsto \text{V} ¦ P \Rightarrow \left\{\ \{\text{f}\} : \text{t} \mapsto \text{V}; \text{F}' : \text{t} \mapsto \text{V} ¦ P', \quad \text{F} : \text{t} \mapsto \text{V} ¦ P''\ \right\}}$$

Mat-Obj-Frame

$$\frac{O_2 ¦ P \Rightarrow \bar{O}}{O_1; O_2 ¦ P \Rightarrow \left\{\ O_1; O_i ¦ P_i \ \middle|\ O_i ¦ P_i \in \bar{O}\ \right\}}$$

$$\boxed{H ¦ P \Rightarrow \wp(H ¦ P)}$$

Mat-Obj

$$\frac{O ¦ P \Rightarrow \bar{O}}{\{\text{a}\} \cdot \langle O \rangle ¦ P \Rightarrow \left\{\ \{\text{a}\} \cdot \langle O_i \rangle ¦ P_i \ \middle|\ O_i ¦ P_i \in \bar{O}\ \right\}}$$

Mat-Addr

$$\frac{\text{A}' \text{ is fresh} \qquad P' = P \wedge \{\text{a}\} \uplus \text{A}' = \text{A} \qquad P'' = P \wedge \{\text{a}\} \cap \text{A} = \emptyset}{\text{A} \cdot \langle O \rangle ¦ P \Rightarrow \left\{\ \{\text{a}\} \cdot \langle O \rangle \ast \text{A}' \cdot \langle O \rangle ¦ P', \quad \text{A} \cdot \langle O \rangle ¦ P''\ \right\}}$$

Mat-Heap-Frame

$$\frac{H_2 ¦ P \Rightarrow \bar{H}}{H_1 \ast H_2 ¦ P \Rightarrow \left\{\ H_1 \ast H_i ¦ P_i \ \middle|\ H_i ¦ P_i \in \bar{H}\ \right\}}$$

$$\boxed{D \Rightarrow D}$$

Mat-Heap

$$\frac{H_2 ¦ P \Rightarrow \bar{H}}{[H_2]_{H_1} ¦ P \Rightarrow \bigvee \left\{\ [H_i]_{H_1} ¦ P_i \ \middle|\ H_i ¦ P_i \in \bar{H}\ \right\}}$$

Mat-Disj

$$\frac{D_2 \Rightarrow D_2'}{D_1 \vee D_2 \Rightarrow D_1 \vee D_2'}$$

Figure 4.6. – Materialization of all of the parts of objects never produces fresh attribute/value trackers. It reuses existing trackers.

one where the object is definitely part of the summary and the other where the object is definitely not part of the summary. In the first case, there are two objects (one singleton, one summary) in the resulting heap, whereas the second case does not perform any materialization. Note that materialization may not lose any precision because the all of the history of how materialization happened is stored in the resulting pure set domain. As a result, with a precise enough domain, materialization can be completely undone. The MAT-HEAP-FRAME is the traditional separation logic frame rule that allows applying some operation (in this case a materialization) to a part of the heap and recombining the result of that with the remaining untouched part of the heap.

The final judgment form has two rules. The first rule, MAT-HEAP delegates to the second judgment form to materialize a heap. This rule is the sole difference between materialization for the two-state abstraction and the single-state abstraction. This rule applies materialization only to the second (current) abstract state and leaves the special (precondition) abstract state alone, not supporting materialization. This rule is responsible for introducing disjunctions in the abstraction for each case split introduced by materialization. Consequently HOO makes extensive use of disjunctions introduced during materialization. The second rule, MAT-DISJ simply handles disjunctions by allowing materialization to be applied to each disjunction separately.

Example 4.6 (Materializing a summary). Consider the following HOO abstraction:

$$[\mathrm{A} \cdot \langle \mathrm{F} : \mathrm{t} \mapsto \mathrm{V} \rangle]_{H_1} \,\dashv\, \{\mathrm{a}\} \subseteq \mathrm{A} \wedge \{\mathrm{f}\} \subseteq \mathrm{F}$$

If the analysis needs to read from a[f], this must be materialized. To achieve the following heap abstraction first the MAT-ADDR rule is applied, then the MAT-ATTR rule is applied to the result, then the rule MAT-VALUE is applied:

$$\begin{bmatrix} \mathrm{A}' \cdot \langle \mathrm{F} : \mathrm{t} \mapsto \mathrm{V} \rangle * \\ \{\mathrm{a}\} \cdot \langle \mathrm{F}' : \mathrm{t} \mapsto \mathrm{V}; \{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\} \rangle \end{bmatrix}_{H_1} \dashv \begin{array}{l} \{\mathrm{a}\} \uplus \mathrm{A}' = \mathrm{A} \\ \wedge \{\mathrm{f}\} \uplus \mathrm{F}' = \mathrm{F} \\ \wedge \{\mathrm{v}\} \subseteq \mathrm{V} \end{array}$$

Materialization operations are designed to split summaries into a singleton and a new summary. When this occurs, the relationship between the singleton, new summary, and old summary is added as a constraint in the set domain. Because the set domain overapproximates, the precise materialization constraint may be lost. Regardless, it is sound to lose this constraint. It simply means that materialization may not be reversed precisely. This is written by saying that if a state and valuation that is in the concretization of a domain D_1, and D_2 can be materialized from D_1, D_2 should contain that state and valuation.

Theorem 1 (Soundness of Materialization). *If $D_1 \Rightarrow D_2$, for all η, σ_1, σ_2, $(\eta, \sigma_1, \sigma_2) \in \gamma(D_1)$ implies that $(\eta, \sigma_1, \sigma_2) \in \gamma(D_2)$.*

4.4. Reading and Writing in Objects

The next part of a definition of an abstract domain is the transfer functions, which determine how interpreting a command k starting from an abstract domain element D yields another abstract domain element D'. The primary operations supported in open-object-focused JavaScript involve reading from and writing to objects. In fact, even direct variable copy (`x = y`), because stack allocated variables are represented as objects are in fact, object reads and writes with the selected attribute being the unit attribute (). This covers the abstract domain transfer functions for reading from and writing to objects.

To unify the presentation of these rules, I introduce a special variety of materialization rule. The rule Mat-Addr-Env (shown below) produces a stack-allocated variable if the variable does not already exist. This means that only a single write operation is required: one that assumes that a particular variable exists and is being overwritten. If the situation occurs where the variable does not already exist, it can be created with any value before immediately being overwritten. Because the program variable x is required to be fresh, necessarily there can only be a single value of x in the heap. This allows this materialization to be applied using an appropriate frame rule. If x were not fresh, the frame rule might not hold [Rey02].

$$\frac{\textsc{Mat-Addr-Env} \qquad \mathtt{x} \text{ is fresh}}{\textsc{Emp} \mathrel{¦} P \Rightarrow \{\mathtt{x}\} \cdot \langle\{()\} : - \mapsto \mathrm{V}\rangle \mathrel{¦} P}$$

The definition of the transfer functions for reading and writing objects is shown in Figure 4.7. The rule T-Single unifies the single-state and the two-state versions of HOO. Because the special, initial state H_1 is simply maintained by the abstraction and all of the operations are applied to the current state H_2, applying the single-state abstract transfer relation to H_2 and then reconstructing the two state abstraction with the result H_2' and the same H_1 gives the expected result.

There are two rules for reading from an object. The first, T-Read-P reads the value contained within an attribute f within the object a and then updates the simple pointer x with the read value. This can only be applied to a singleton object, attribute, and value, so appropriate materializations must have been applied before applying this rule. The second rule, T-Read-N reads the default, undef value when the attribute does not exist within the object A. In order to apply this rule, materialization must have added the constraint that f is not in the object O. This is proven through the exclusion rules.

The rule for writing, T-Write assumes that a target object at address a exists and then writes to that object. The write has two critical parts: the overwrite and the extension. These parts are shown in Figure 4.8. The overwrite modifies the target object O producing a new object O' that no longer has the written attribute f in it. This works through the overwriting rules described below. The extension part of the rule adds a new partition with the singleton attribute f as its sole constituent and the corresponding value

T-SINGLE

$$\frac{[H_2 \mathrel{¦} P] \;\; k \;\; [H_2' \mathrel{¦} P']}{\left[[H_2]_{H_1} \mathrel{¦} P\right] \;\; k \;\; \left[[H_2']_{H_1} \mathrel{¦} P'\right]}$$

$$\boxed{\left[[H_2]_{H_1} \mathrel{¦} P\right] \;\; k \;\; \left[[H_2']_{H_1} \mathrel{¦} P'\right]}$$

$$\boxed{[H \mathrel{¦} P] \;\; k \;\; [H' \mathrel{¦} P']}$$

T-READ-P

$$\frac{}{\begin{array}{c}[\mathrm{x} \mapsto \mathrm{V_x} * \mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle \{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O \rangle \mathrel{¦} P] \\ \mathtt{x = y[z]} \\ [\mathrm{x} \mapsto \{\mathrm{v}\} * \mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle \{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O \rangle \mathrel{¦} P]\end{array}}$$

T-READ-N

$$\frac{P \vdash \{\mathrm{f}\} \not\subseteq O}{\begin{array}{c}[\mathrm{x} \mapsto \mathrm{V_x} * \mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle O \rangle \mathrel{¦} P] \\ \mathtt{x = y[z]} \\ [\mathrm{x} \mapsto \{\mathsf{undef}\} * \mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle O \rangle \mathrel{¦} P]\end{array}}$$

T-WRITE

$$\frac{O \setminus \{\mathrm{f}\} \mathrel{¦} P \Rightarrow O' \mathrel{¦} P'}{\begin{array}{c}[\mathrm{x} \mapsto \{\mathrm{a}\} * \mathrm{y} \mapsto \{\mathrm{f}\} * \mathrm{z} \mapsto \{\mathrm{v}\} * \{\mathrm{a}\} \cdot \langle O \rangle \mathrel{¦} P] \\ \mathtt{x[y] = z} \\ \left[\mathrm{x} \mapsto \{\mathrm{a}\} * \mathrm{y} \mapsto \{\mathrm{f}\} * \mathrm{z} \mapsto \{\mathrm{v}\} * \{\mathrm{a}\} \cdot \langle O'; \{\mathrm{f}\} : - \mapsto \{\mathrm{v}\} \rangle \mathrel{¦} P'\right]\end{array}}$$

T-COPY-P

$$\frac{O_1 \setminus \{\mathrm{f}\} \mathrel{¦} P \Rightarrow O_1' \mathrel{¦} P'}{\begin{array}{c}\left[\begin{array}{c}\mathrm{x} \mapsto \{\mathrm{a_1}\} * \mathrm{y} \mapsto \{\mathrm{f}\} * \mathrm{z} \mapsto \{\mathrm{a_2}\} * \\ \{\mathrm{a_1}\} \cdot \langle O_1 \rangle * \{\mathrm{a_2}\} \cdot \langle \{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O_2\end{array} \rangle \mathrel{¦} P\right] \\ \mathtt{x[y] = z[y]} \\ \left[\begin{array}{c}\mathrm{x} \mapsto \{\mathrm{a_1}\} * \mathrm{y} \mapsto \{\mathrm{f}\} * \mathrm{z} \mapsto \{\mathrm{a_2}\} * \\ \{\mathrm{a_1}\} \cdot \langle O_1'; \{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\} \rangle * \{\mathrm{a_2}\} \cdot \langle \{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O_2\end{array} \rangle \mathrel{¦} P'\right]\end{array}}$$

Figure 4.7. – Reading and writing objects

$$\boxed{P \vdash \{\mathrm{f}\} \not\subseteq O}$$

$$\textsc{Exc-Sep}\ \frac{P \vdash \{\mathrm{f}\} \not\subseteq O_1 \qquad P \vdash \{\mathrm{f}\} \not\subseteq O_2}{P \vdash \{\mathrm{f}\} \not\subseteq O_1; O_2} \qquad \textsc{Exc-None}\ \frac{}{P \vdash \{\mathrm{f}\} \not\subseteq \textsc{None}} \qquad \textsc{Exc-Part}\ \frac{P \vdash \{\mathrm{f}\} \cap \mathrm{F} = \emptyset}{P \vdash \{\mathrm{f}\} \not\subseteq \mathrm{F} : \mathrm{t} \mapsto \mathrm{V}}$$

$$\boxed{O \setminus \{\mathrm{f}\} \,¦\, P \Rightarrow O' \,¦\, P'}$$

$$\textsc{Over-Sep}\ \frac{O_1 \setminus \{\mathrm{f}\} \,¦\, P \Rightarrow O'_1 \,¦\, P' \qquad O_2 \setminus \{\mathrm{f}\} \,¦\, P' \Rightarrow O'_2 \,¦\, P''}{O_1; O_2 \setminus \{\mathrm{f}\} \,¦\, P \Rightarrow O'_1; O'_2 \,¦\, P''} \qquad \textsc{Over-None}\ \frac{}{\textsc{None} \setminus \{\mathrm{f}\} \,¦\, P \Rightarrow \textsc{None} \,¦\, P}$$

$$\textsc{Over-Part}\ \frac{P' = P \wedge \mathrm{F}' = \mathrm{F} \setminus \{\mathrm{f}\} \qquad \mathrm{F}' \text{ is fresh}}{\mathrm{F} : \mathrm{t} \to \mathrm{V} \setminus \{\mathrm{f}\} \,¦\, P \Rightarrow \mathrm{F}' : \mathrm{t} \to \mathrm{V} \,¦\, P'}$$

Figure 4.8. – Reading and writing objects overwrite and extension

for the write. Note that the write operation does not propagate attribute/value trackers. To do that requires the use of the copy operation.

Transferring attribute/value trackers Attribute/value trackers are transferred from one object to another by assignment. For simplicity, I assume here that all assignments between objects are transformed into the form of a simultaneous read from an object and a write to another object. When the attribute being read and written matches so that an attribute/value pair is being copied, there is an opportunity to transfer that attribute/value pair from one object to the other. When this transfer happens, the attribute/value tracker can be transferred as well.

The T-Copy-P rule in Figure 4.7 shows the of the transfer relation that enables an attribute/value tracker transfer. Note that it only applies when the attribute being read exists within the object. If it does not exist, the default, unknown attribute will be retrieved and thus does not have an attribute/value tracker associated with it. In this case, a sequence of T-Read-N and T-Write achieves the same goal. When the attribute does exist in the source object a_2 and a tracker t exists for that attribute, when that attribute is copied to the first object a_1, the tracker t is also copied because the particular attribute/value relationship can remain across a copy operation.

Introducing attribute/value trackers Attribute/value trackers should be introduced at chosen program points where the first of the paired states is selected. For example, when constructing an initial abstract state it would be normal to express it as $[H]_H \,¦\, P$ where the two described heaps are identical. In this instance, fresh attribute/value trackers should be introduced for each partition in H. This establishes the initial relationship between

the initial abstract state and the current abstract state and then any attribute/value trackers that are preserved strengthen the relationship between the two states.

Additionally, attribute/value trackers can be introduced at other times. The benefits of doing so are less significant as freshly introduced trackers cannot relate objects from one time to another, but instead are limited to relating multiple objects in the same time. However, as trackers are incomparable unless they are syntactically equal, freely introducing fresh trackers can prevent inclusion checking from succeeding and thus prevent the analysis from terminating. To avoid this problem in two-state HOO, one acceptable strategy is to not introduce trackers after the pre-condition of the analysis. This ensures that the analysis can terminate while still preserving the key benefit of relating objects in the pre-condition to objects in the current state.

Excluding attributes Attribute exclusion checks the pure set constraints P to ensure that a given attribute f is not in an object. It does this through three rules Exc-Sep, Exc-None, and Exc-Part. The purpose of the first two rules is to allow the third rule to be applied to every partition within an object O. The third rule checks that $\{\mathrm{f}\} \nsubseteq \mathrm{F}$ by checking if P implies that the constraint that these two sets are disjoint, which works because the attribute is always a singleton attribute.

Overwriting attributes Overwriting attributes works similarly to excluding attributes. It produces a fresh object and pure set domain where each partition is constrained to be the same as the old partition minus the particular attribute that is being overwritten. There are three rules to accomplish this. The first, Over-Sep applies overwriting to each sub-partition. If an empty partition is found, nothing happens as described by Over-None. Finally, if a base-level partition is found, a fresh attribute set F' is introduced that is constrained to be equal to the original attribute set F minus the singleton set $\{\mathrm{f}\}$. The resulting object no longer has f as one of its attributes.

Theorem 2 (Soundness of Transfer Functions). *Transfer functions are sound because:*

$$\forall k, \sigma_0, \sigma, \sigma', D, D'.\ \langle\sigma\rangle\, k\, \langle\sigma'\rangle \text{ and } [D]\ k\ [D'] \text{ and } \exists\eta.\ (\eta, \sigma_0, \sigma) \in \gamma(D) \text{ implies } \exists\eta'.\ (\eta', \sigma_0, \sigma') \in \gamma(D')$$

4.5. Automatic Invariant Inference

In this section I give the join, widening, and inclusion check algorithms that are required for automatically and soundly generating loop invariants. Here the focus is inferring loop invariants for **`for-in`** loops — the primary kind of loop for object-manipulation. In effect, **`for-in`** loops are interpreted as a more traditional while loop that iterates over the elements within the attribute sets in the object that is being iterated over. Critical to this process is the introduction of iteration progress variables F_{o} and F_{i} to keep track of which attributes of the object have been visited over the course of the iteration and those that have not.

The analysis of **for-in** loops first translates these loops into **while** loops. This allows HOO to follow the standard abstract interpretation procedure for loops, while introducing iteration-progress variables to aid the analysis in inferring precise loop invariants. The translation of `for(x in y) { k }` uses several additional commands not defined as part of the open-object-focused JavaScript:

```
Fi = attr(y);
Fo = ∅;
while(Fi ≠ ∅) {
  x = choose(Fi);
  Fi = Fi \ {x};
  k
  Fo = Fo ∪ {x};
}
```

The `attr` command constrains the $\mathsf{F_i}$ set to be equal to the union of all of the partitions of the object, while assigning $\mathsf{F_o}$ to the empty set constrains it to be initially empty. Then on each iteration of the loop, an element is selected from the $\mathsf{F_i}$ set, bound to `x`, and then removed from the $\mathsf{F_i}$ set. The original body of the for loop is then executed. Finally, the selected element `x` is added to $\mathsf{F_o}$. The purpose of this iterative process is to allow relationships between objects to iteratively form. Initially there may be no relationships between objects, but if the attribute `x` is copied to another object, that makes a relationship between those two objects on just that attribute `x`. At the end of the loop body that attribute `x` is folded into $\mathsf{F_o}$, which happens in all objects where `x` occurs. This makes the relationship between objects over all of $\mathsf{F_o}$, which then iteratively grows until it is equal to the full set of attributes of the object pointed to by `y`.

These iteration-progress variables are essential for performing strong updates. When analyzing an iteration of a loop, partitions that arise from attribute materialization arise simultaneously with partitions that arise in iteration-progress variables. Thus, these partitions become related and even when partitions from attribute materialization must be summarized, the relationship with the iteration progress variable is maintained. The summarization process occurs as part of join and widening.

Join Algorithm: The join algorithm takes two abstract states D_1 and D_2 and computes an overapproximation of all program states described by each of these abstract states. The rules for the join algorithm are shown in Figure 4.9. When joining disjunctions, either the join can be converted to another disjunction via the JOIN-DISJ rule, or appropriate pairs of disjuncts can be matched and joined via the JOIN-UPPER rule. When joining two heaps as in the JOIN-HEAP rule, the two heaps require matching initial heaps H_0, but then the current heaps are joined via the heap join rules JOIN-EMP and JOIN-OBJ. When joining heap abstractions, the algorithm must match objects in H_1 and objects in H_2 to objects in a resulting abstract memory H_3. This matching of objects can be described by two mapping functions M_1 and M_2, where $M_1 : \overline{\mathrm{Symbol}}_1 \overset{\mathrm{fin}}{\to} \overline{\mathrm{Symbol}}_3$ maps symbols from H_1 to symbols from H_3 and $M_2 : \overline{\mathrm{Symbol}}_2 \overset{\mathrm{fin}}{\to} \overline{\mathrm{Symbol}}_3$ maps symbols from H_2 to symbols from H_3. However, because HOO abstracts open objects, the join

$$\boxed{D_1 \sqcup D_2 \rightsquigarrow D_3}$$

$$\textsc{Join-Disj}\quad \frac{}{D_1 \vee D_2 \sqcup D_3 \vee D_4 \rightsquigarrow D_1 \vee D_3 \vee D_2 \vee D_4}$$

$$\textsc{Join-Upper}\quad \frac{D_1 \sqcup D_3 \rightsquigarrow D_5 \qquad D_2 \sqcup D_4 \rightsquigarrow D_6}{D_1 \vee D_2 \sqcup D_3 \vee D_4 \rightsquigarrow D_5 \vee D_6}$$

$$\textsc{Join-Heap}\quad \frac{\begin{array}{c} M_{1_I}, M_{2_I}, \{\} \vdash H_1 \mathbin{\dot{|}} P_1 \sqcup H_2 \mathbin{\dot{|}} P_2 \rightsquigarrow H_3 \mathbin{\dot{|}} P_3 \\ M_{1_I} = \bigcup_{\mathrm{A}_1 \in \mathrm{roots}(H_1)} [\mathrm{A}_1 \mapsto \mathrm{A}_1] \qquad M_{2_I} = \bigcup_{\mathrm{A}_2 \in \mathrm{roots}(H_2)} [\mathrm{A}_2 \mapsto \mathrm{A}_2] \end{array}}{[H_1]_{H_0} \mathbin{\dot{|}} P_1 \sqcup [H_2]_{H_0} \mathbin{\dot{|}} P_2 \rightsquigarrow [H_3]_{H_0} \mathbin{\dot{|}} P_3}$$

$$\boxed{M_1, M_2, P_J \vdash H_1 \mathbin{\dot{|}} P_1 \sqcup H_2 \mathbin{\dot{|}} P_2 \rightsquigarrow H_3 \mathbin{\dot{|}} P_3}$$

$$\textsc{Join-Emp}\quad \frac{M_1, M_2, P_J \vdash P_1 \sqcup P_2 \rightsquigarrow P_3}{M_1, M_2, P_J \vdash \textsc{Emp} \mathbin{\dot{|}} P_1 \sqcup \textsc{Emp} \mathbin{\dot{|}} P_2 \rightsquigarrow \textsc{Emp} \mathbin{\dot{|}} P_3}$$

$$\textsc{Join-Obj}\quad \frac{\begin{array}{c} M_1' = [\mathrm{A}_1 \mapsto \mathrm{A}_3] \cup M_1 \\ M_2' = [\mathrm{A}_2 \mapsto \mathrm{A}_3] \cup M_2 \qquad M_1', M_2', P_J \vdash O_1, P_1 \sqcup O_2, P_2 \rightsquigarrow O_3, M_1'', M_2'', P_J' \\ M_1'', M_2'', P_J' \vdash H_1 \mathbin{\dot{|}} P_1 \sqcup H_2 \mathbin{\dot{|}} P_2 \rightsquigarrow H_3 \mathbin{\dot{|}} P_3 \end{array}}{M_1, M_2, P_J \vdash \mathrm{A}_1 \cdot \langle O_1 \rangle * H_1 \mathbin{\dot{|}} P_1 \sqcup \mathrm{A}_2 \cdot \langle O_2 \rangle * H_2 \mathbin{\dot{|}} P_2 \rightsquigarrow \mathrm{A}_3 \cdot \langle O_3 \rangle * H_3 \mathbin{\dot{|}} P_3}$$

Figure 4.9. – Rules for joining two heaps. The Join-Disj rule permits joining by disjunction, which should be omitted for widening. The Join-Obj rule applies the templates given in Table 4.1 to join two objects together.

algorithm must match partitions of objects as well. This matching is represented with a relation $P_J \subseteq \wp(\mathrm{F}_1) \times \wp(\mathrm{F}_2) \times \mathrm{F}_3$ that relates sets of partitions from objects in H_1 and H_2 to partitions in H_3. However, join can only be completed if the initial heap H_0 matches between the two abstract states. Because partitions can be split and because new, empty partitions can be created, join can produce an unbounded number of partitions.

The fundamental challenge for the HOO abstraction's join algorithm is computing these symbol matchings M_1, M_2, and P_J. To construct the matchings, in the Join-Heap rule, the join algorithm begins at the symbolic addresses of stack allocated variables found via roots(). It adds equivalent variables from the three graphs to M_1 and M_2, and then it begins an iterative process. The Join-Obj rule starts from a matching that already exists in M_1 and M_2; it derives additional matchings that are potential consequences. To derive these additional matchings, a template system is used. The templates consume corresponding parts of a memory abstraction, producing a resultant memory abstraction that holds under the matchings. This iterative process is applied until no more templates

$$\dfrac{[], [], \{\} \vdash H_1 \mid P \sqcup H_2 \mid P \rightsquigarrow H_3 \mid P'}{H_1 * H_2 \mid P \Rightarrow H_3 \mid P'} \textsc{ Weaken-Heap} \qquad \dfrac{D_1 \sqcup D_2 \rightsquigarrow D_3}{D_1 \vee D_2 \Rightarrow D_3} \textsc{ Weaken-Disj}$$

Figure 4.10. – Rules for weakening heaps prior to join. These unify the heaps so that object can be appropriately matched or so that disjuncts can be appropriately matched.

Table 4.1. – Join templates match objects in two abstract heaps, producing a third heap that overapproximates both. Matchings M_1, M_2, P_J are generated on the fly and used in the set domain join after the heaps are joined.

Prerequisites	H_1, P_1 $\sqcup$ H_2, P_2 $\rightsquigarrow$	Result
$M_1(\mathrm{A}_1) = \mathrm{A}_3$ $M_2(\mathrm{A}_2) = \mathrm{A}_3$	[A_1: $\mathrm{F}_1 \mid \mathrm{V}'_1$] $\sqcup$ [A_2: $\mathrm{F}_2 \mid \mathrm{V}'_2$] $\rightsquigarrow$ [A_3: $\mathrm{F}_3 \mid \mathrm{V}'_3$]	$M_1(\mathrm{V}'_1) = \mathrm{V}'_3, \quad M_2(\mathrm{V}'_2) = \mathrm{V}'_3$ $(\{\mathrm{F}_1\}, \{\mathrm{F}_2\}, \mathrm{F}_3) \in P_J$
$M_1(\mathrm{A}_1) = \mathrm{A}_3$ $M_2(\mathrm{A}_2) = \mathrm{A}_3$ remainder of object matches	[A_1: $\vdots \mid \vdots$; $\mathrm{F}^i_1 \mid \mathrm{V}^i_1$; $\vdots \mid \vdots$; $\mathrm{F}^m_1 \mid \mathrm{V}^m_1$; $\vdots \mid \vdots$] $\sqcup$ [A_2: $\vdots \mid \vdots$; $\mathrm{F}^j_2 \mid \mathrm{V}^j_2$; $\vdots \mid \vdots$; $\mathrm{F}^n_2 \mid \mathrm{V}^n_2$; $\vdots \mid \vdots$] $\rightsquigarrow$ [A_3: $\vdots \mid \vdots$; $\mathrm{F}^k_3 \mid \mathrm{V}^k_3$; $\vdots \mid \vdots$]	$(\{\mathrm{F}^i_1, \cdots, \mathrm{F}^m_1\}, \{\mathrm{F}^j_2, \cdots \mathrm{F}^n_2\}, \mathrm{F}^k_3) \in P_J$ $M_1(\mathrm{V}^i_1) = \mathrm{V}^k_3, \quad M_2(\mathrm{V}^j_2) = \mathrm{V}^k_3$ $\vdots \qquad \vdots$ $M_1(\mathrm{V}^m_1) = \mathrm{V}^k_3, \quad M_2(\mathrm{V}^n_2) = \mathrm{V}^k_3$

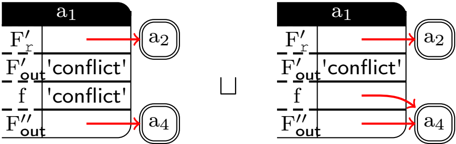
can be applied. It is assumed that all heap will be matched, however an alternative is to add TRUE to the result for any unmatched parts. The result of join is complete matchings M_1, M_2, and P_J, as well as, a memory abstraction H_3. To get the resulting set abstraction P_3, the sets are joined under the same matchings, where multiple matchings are interpreted as a union.

To complete the join process, it is required that disjuncts and object summaries are matched one to one. To ensure that this happens, weakening rules are applied. These rules are shown in Figure 4.10. The WEAKEN-HEAP rule merges two parts of the heap into a single heap by using rules for join within a heap. Similarly the WEAKEN-DISJ rule merges two disjuncts using a join rule so that disjuncts can be matched pair-wise in the JOIN-UPPER rule.

There are two templates described in Table 4.1. The first template joins any two objects that have only one partition. The values from that partition are added to the mapping as well as the default values. The second template is parametric. If some number of partitions can be matched with some number of partitions then those can all be merged into a single partition in the result. This template requires applying other rules to complete the joining of the objects. If it is unknown how to match partitions for all of an object, this template allows matching part of the object. If the result is that remaining partitions are single partitions, even if there is no natural way to match them, they will be matched by applying template one.

Example 4.7 (Joining objects). Here, two a_1 objects that are extracted from an

example implementing traits as in traits.js are joined. The objects to be joined look like the following:



The join constructs matchings M_1, M_2, and P_J. Initially $M_1 = [\mathrm{a}_1 \mapsto \mathrm{a}_1]$, $M_2 = [\mathrm{a}_1 \mapsto \mathrm{a}_1]$, and $P_J = \emptyset$. If $\mathrm{F}'_{\mathsf{out}}$ were matched with $\mathrm{F}'_{\mathsf{out}}$ or $\mathrm{F}''_{\mathsf{out}}$ were matched with $\mathrm{F}''_{\mathsf{out}}$, the result would be an imprecise join because it would be forced to match f with itself even though it has two values that should not be joined. Instead, the second template can be applied to merge partitions with like values, thus merging f with $\mathrm{F}'_{\mathsf{out}}$ in first object and with $\mathrm{F}''_{\mathsf{out}}$ in second object. Since the only remaining partition is F'_{r}, F'_{r} and F'_{r} are matched giving the following matchings and join result:

$$M_1 = [\mathrm{a}_1 \mapsto \mathrm{a}_1, \mathrm{a}_2 \mapsto \mathrm{a}_2, \mathrm{a}_4 \mapsto \mathrm{a}_4]$$
$$M_2 = [\mathrm{a}_1 \mapsto \mathrm{a}_1, \mathrm{a}_2 \mapsto \mathrm{a}_2, \mathrm{a}_4 \mapsto \mathrm{a}_4]$$
$$P_J = \{(\{\mathrm{F}'_{\mathrm{r}}\}, \{\mathrm{F}'_{\mathrm{r}}\}, \mathrm{F}'_{\mathrm{r}}),\ (\{\mathrm{F}'_{\mathsf{out}}, \mathrm{f}\}, \{\mathrm{F}'_{\mathsf{out}}\}, \mathrm{F}'_{\mathsf{out}}),\ (\{\mathrm{F}''_{\mathsf{out}}\}, \{\mathrm{F}''_{\mathsf{out}}, \mathrm{f}\}, \mathrm{F}''_{\mathsf{out}})\}$$

In the implementation, join is implemented using the JOIN-DISJ rule which introduces a case split for each join. To maximize precision, this works well. Only when widening are the other join rules used.

Widening algorithm: In HOO, the join and widening algorithms are nearly identical. However, unlike join, widening must select matchings that ensure convergence of the analysis, by guaranteeing that the number of disjuncts and the number of partitions do not grow unboundedly and that the arrangement of the disjuncts and partitions are ultimately fixed (i.e. there is no oscillation in which partitions are matched during widening). While there are many possible approaches that meet these criteria, the implementation of HOO utilizes allocation site information to resolve decisions during the matching process. Only objects from the same allocation site may be matched, which causes only attribute sets whose corresponding values are from the same allocation site to be matched. To ensure convergence, after some number of iterations, all objects from the same allocation site can be forced to be matched. This bounds the partitions per abstract object to one per allocation site and bounds the number of abstract objects to one per allocation site, so as long as the underlying set domain converges on an abstraction for each partition, the analysis will converge.

The implementation of widening constructs initial mappings using the allocation site information retrieved for an address or value symbol using the alloc-id () function. A new

symbol for each used allocation site is produced using the symbol () function. The initial mappings are defined as

$$M_1 = \bigcup_{\mathrm{A} \in \mathrm{Dom}(H_1)} [\mathrm{A} \mapsto \mathrm{symbol}\,(\text{alloc-id}\,(\mathrm{A}))]$$

$$M_2 = \bigcup_{\mathrm{A} \in \mathrm{Dom}(H_2)} [\mathrm{A} \mapsto \mathrm{symbol}\,(\text{alloc-id}\,(\mathrm{A}))]$$

where Dom (H) retrieves all of the address symbols in a portion of a heap H. Then in the application of the object joining rules, the second rule in Table 4.1 is applied for each different allocation site for each value.

Theorem 3 (Join Soundness). *Join is sound under matchings M_1, M_2, P_J because*

$$\begin{aligned}
&\textit{if } P \vdash [H_1]_{H_0} \,¦\, P_1 \sqcup [H_2]_{H_0} \,¦\, P_2 \rightsquigarrow [H_3]_{H_0} \,¦\, P_3 \textit{ then} \\
&\quad \forall \sigma_0, \sigma, \eta_1, \eta_2.\ (\eta_1, \sigma_0, \sigma) \in \gamma([H_1]_{H_0} \,¦\, P_1) \vee (\eta_2, \sigma_0, \sigma) \in \gamma([H_2]_{H_0} \,¦\, P_2) \\
&\qquad \wedge \forall (\bar{\mathrm{V}}_1, \bar{\mathrm{V}}_2, \mathrm{V}_3) \in P.\ \bigcup \{\, \eta_1(\mathrm{V}_1) \mid \mathrm{V}_1 \in \bar{\mathrm{V}}_1 \,\} = \bigcup \{\, \eta_2(\mathrm{V}_2) \mid \mathrm{V}_2 \in \bar{\mathrm{V}}_2 \,\} \Rightarrow \\
&\qquad\quad \exists \eta_3.\ (\eta_3, \sigma_0, \sigma) \in \gamma([H_3]_{H_0} \,¦\, P_3) \\
&\qquad\qquad \wedge \forall (\bar{\mathrm{V}}_1, \bar{\mathrm{V}}_2, \mathrm{V}_3) \in P.\ \bigcup \{\, \eta_1(\mathrm{V}_1) \mid \mathrm{V}_1 \in \bar{\mathrm{V}}_1 \,\} = \eta_3(\mathrm{V}_3)
\end{aligned}$$

where P is a uniform, combined version of M_1, M_2, and P_J and is defined as

$$P \stackrel{def}{=} \left\{ (\bar{\mathrm{V}}_1, \bar{\mathrm{V}}_2, \mathrm{V}_3) \;\middle|\; \begin{array}{l} \mathrm{V}_3 \in \mathrm{Codom}\,(M_1) \cup \mathrm{Codom}\,(M_2) \\ \wedge\, \bar{\mathrm{V}}_1 = \{\, \mathrm{V}_1 \mid M_1(\mathrm{V}_1) = \mathrm{V}_3 \,\} \\ \wedge\, \bar{\mathrm{V}}_2 = \{\, \mathrm{V}_2 \mid M_1(\mathrm{V}_2) = \mathrm{V}_3 \,\} \end{array} \right\} \cup P_J$$

The purpose of the soundness theorem is to state that not only does every single concrete state that is in the concretization of both $[H_1]_{H_0} \,¦\, P_1$ and $[H_2]_{H_0} \,¦\, P_2$, occur in the concretization of $[H_3]_{H_0} \,¦\, P_3$, but also there is a relationship between the valuations η_1, η_2, and η_3.

Properties other than soundness are not stated due to the dependence of HOO's behavior on its instantiation. Because of the non-trivial interaction between the set domain and HOO, properties of HOO are affected by properties of the set domain. More precise set domain operations typically yield more precision in HOO. Additionally, the choice of heuristics for template application can affect the results of join, widening, and inclusion check, thus leading to a complex dependency between precision and heuristics. While this dependence can affect many properties, it does not affect soundness.

Inclusion Check Algorithm: Inclusion checking determines if an abstract state is already described by another abstract state. The process for deciding if an inclusion holds is similar to the join processes and is described formally in Chapter C. If $M, P_I \vdash [H_a]_{H_0} \,¦\, P_a \sqsubseteq [H_b]_{H_0} \,¦\, P_b$, all pairs of concrete states described by $[H_a]_{H_0} \,¦\, P_a$ must be contained in the set of all pairs of concrete states described by $[H_b]_{H_0} \,¦\, P_b$. It works in a fashion similar to join by constructing matchings M and P_I from symbols in H_a, P_a to symbols in H_b, P_b. It employs the same methodology as join. Objects are matched,

one-by-one, until no more matches can be made. This matching builds up the mapping M that is then used for an inclusion check in the set domain. If the mapping was successfully constructed and the inclusion check holds in the set domain, the inclusion check holds on the HOO domain. The templates for augmenting the mapping are essentially the same as those for join shown in Table 4.1, except with only M_1 and with P_I only using the first and third components and where H_2, P_2 is ignored with H_1, P_1 corresponding to H_a, P_a and the result corresponding to H_b, P_b.

Theorem 4 (Inclusion Soundness). *Inclusion checking is sound under matchings M, P_I because assuming that P is defined as follows:*

$$P \stackrel{def}{=} \left\{ (\bar{\mathrm{V}}_a, \mathrm{V}_b) \;\middle|\; \mathrm{V}_b \in \mathrm{Codom}\,(M) \wedge \bar{\mathrm{V}}_a = \{\, \mathrm{V}_a \mid M(\mathrm{V}_a) = \mathrm{V}_b \,\} \right\} \cup P_I$$

If $M, P_I \vdash H_a, P_a \sqsubseteq H_b, P_b$ *then* $\forall \eta_a, \sigma_0, \sigma.\ (\eta_a, \sigma_0, \sigma) \in \gamma([H_a]_{H_0} \,\vdots\, P_a) \Rightarrow$
$\exists \eta_b.\ (\eta_b, \sigma_0, \sigma) \in \gamma([H_b]_{H_0} \,\vdots\, P_b) \wedge \forall (\bar{\mathrm{V}}_a, \mathrm{V}_b) \in P.\ \bigcup \left\{ \eta_a(\mathrm{V}_a) \;\middle|\; \mathrm{V}_a \in \bar{\mathrm{V}}_a \right\} = \eta_b(\mathrm{V}_b)$

4.6. Related Work

Analyses for dynamic languages: Because one of the main features of dynamic languages is open objects, all analyses for dynamic languages must handle open objects to a degree. As opposed to directly abstracting open objects, TAJS [JMT10, JMT09], WALA [SDC+12], JSAI [HWCK14, KSW+13], and SAFE [LWJ+12, BCLR14] extend standard field-sensitive analyses to JavaScript by adding a summary field for all unknown attributes. They employ clever interprocedural analysis tricks to propagate statically known object attributes through loops and across function call boundaries. Consequently, with the whole program, they can often precisely verify properties of open-object manipulating programs. Without the whole program, these techniques lose precision because they conflate all unknown object attributes into a single summary field and weakly update it through loops.

Type systems for dynamic languages: There has been a recent push to add more types support to dynamic languages. For example TypeScript [BAT14] adds a gradual type system [ST07] to JavaScript. Similarly, there are now type systems for Python [VKSB14], Ruby [FAFH09], and Scheme [TF08]. All of these type systems support extensible records to a degree. The Python type system supports them through a variety of flow-sensitivity and monotonic objects. The Ruby system supports them through record combinators, such as explicit support for mixins. The Scheme system supports extensible records through a form subtyping. Additionally, some static languages support extensible records through row-polymorphism [Rém89]. Regardless, they all lack the ability to use complex hand-written loops to manipulate objects, freely adding and removing attributes all while keeping track of where attributes and values originated. Furthermore, many type systems for dynamic languages (such as the Ruby one) elect to be unsound, thus both raising errors where there are none and not raising errors when they exist. Additionally, type systems typically try to avoid reasoning explicitly about the heap.

Local heap analyses: Much of the work on HOO has been inspired by a plethora of work on separation-logic-based analysis. Over the years this work has ranged from inclusion checking with fixed inductive definitions [BCO05], to invariant inference with fixed inductive definitions [BCC+07, BCI11], to invariant inference with user-specified inductive definitions [CR08, Cha08], to modular analysis with fixed inductive definitions [CDOY11]. HOO does not currently incorporate inductive definitions. It uses set-based summaries to represent unbounded data structures. The inspiration comes in the form of local reasoning. Local reasoning allows framing, which allows the definition of the behavior of operations only on the portion of memory that can be affected by the operation. Further, HOO's inference of partitions of objects is related to inferring inductive definitions because inferring the relational structure between an unbounded number of objects is the essence of the inductive definition inference problem.

Analyses for containers: Because objects in dynamic languages behave similarly to containers, it is possible that a container analysis could be adapted to this task. Powerful container analyses such as [DDA11] and [GMT08] can represent and infer arbitrary partitions of containers. This is similar to HOO except that they do not use set abstractions to represent the partitions, but instead use SMT formulas and quantifier templates. For some applications these are excellent choices, but for dynamic languages where the key type of the containers is nearly always strings, this suffers. HOO can use abstract domains for sets [CCS13, PTTC11] and thus if these domains are parametric over their value types, HOO can support nearly any key-type abstraction.

Arrays and lists are restricted forms of containers on which there has been a significant amount of work [CCL11, KV09, HP08, JM07, GRS05, DDA10, BDES12]. The primary difference between arrays and more general containers and open objects is that arrays typically contain related values next to one another. Partitions of arrays are implicitly ordered and because array keys typically do not have gaps, partitions are defined using expressions that identify partition boundaries. Because open objects have gaps and are unordered, array analyses are not applicable. Regardless, array abstraction inspires the partitioning of open objects that we use.

Decision procedures: In addition there are analyses that do not handle loops without annotations for both dynamic languages and containers. DJS [CHJ12, CRJ12] is a flow-sensitive dependent type system for JavaScript. It can infer intermediate states in straight-line code, but it requires annotations for loops and functions. Similarly JuS [GMS12] supports straight-line code for JavaScript. Jahob and its brethren [Kun07] use a battery of different decision procedures to analyze containers and the heap together for Java programs. Finally, array decision procedures [dMB09, BMS06] can be adapted to containers, but all of these approaches require significant annotation of non-trivial loop invariants to be effective on open-object-manipulating programs.

4.7. Heap with Open Objects Summary

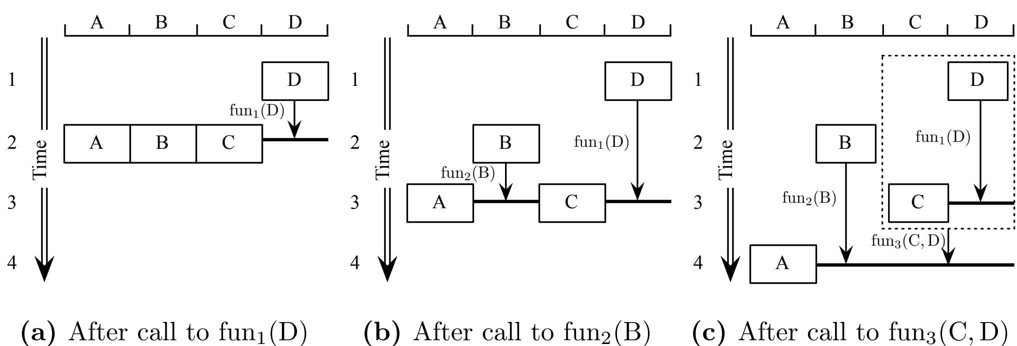
The HOO abstract domain is capable of automatically inferring facts about object manipulating programs even when objects are completely unknown. This is due to HOO's

use of multiple partitions where each is named with a unique symbol that is tracked externally in a relational domain for sets. Therefore these symbols can be related to one another. Critically HOO achieves strong updates by materializing partitions on the fly and by using iteration progress sets to incrementally establish relationships between multiple objects. HOO exists in both a single-state and two-state version, where the two-state version can be used to infer function summaries. Further, due to the use of attribute/value trackers, two-state HOO can be completely precise for programs that solely rely upon copying attributes and values from one object to another, even if the objects were completely unknown at the start.

5. Function Abstraction: Desynchronized Separation

JavaScript libraries do not only take objects as inputs. It is common for inputs to include callback functions as parameters. These parameters cause problems for static analysis because when these callback functions are called, the effects are unknown with respect to the library. Desynchronized separation is a mechanism for keeping track of assumptions about the scope of effects when calling callbacks.

Desynchronization works by representing different parts of the heap at different points in the execution. In effect, parts of the abstract state are desynchronized from other parts of the abstract state. This allows regions of the heap that may have been affected by a callback whose behavior is unknown to be halted in the analysis at the point when the callback occurs. When this desynchronization happens, the portion of the heap that may be affected by the callback is locked in time, but is tagged with enough information to resynchronize it with the rest of the heap should the need arise.



(a) After call to $\mathrm{fun}_1(\mathrm{D})$ **(b)** After call to $\mathrm{fun}_2(\mathrm{B})$ **(c)** After call to $\mathrm{fun}_3(\mathrm{C}, \mathrm{D})$

Figure 5.1. – Three separate desynchronizations after calling three successive functions on four regions of memory. In (c) A is the current analysis state where as regions B, C, and D have all been desynchronized. The D region has been desynchronized twice.

Example 5.1 (Desynchronization). To demonstrate the power of desynchronization, Figure 5.1 shows the process pictorially. The program being considered has four separate regions of memory A, B, C, and D that can be identified by some analysis and the program is about to evaluate three function calls whose bodies are unknown in sequence: $\mathrm{fun}_1(\mathrm{D})$; $\mathrm{fun}_2(\mathrm{B})$; $\mathrm{fun}_3(\mathrm{C}, \mathrm{D})$. Figures 5.1 (a), (b), and (c) show the state of desynchronization after each of these calls. Initially, at time 1, all memory is synchronized and represented at time 1.

When analyzing the call to $\mathrm{fun}_1(\mathrm{D})$, which can only affect region D, the body is unknown and thus the analysis cannot continue. However, because the function can only affect the memory region D, it is possible to proceed if the heap is desynchronized. The result of the desynchronization is shown in Figure 5.1a. Regions A, B, and C are allowed to proceed on to time 2, but region D stays locked at time 1 and becomes inaccessible. This inaccessibility is critical because any of that memory in region D may have been mutated by the call to $\mathrm{fun}_1(\mathrm{D})$, and without any knowledge of what fun_1 did, it is impossible to say what the effect of accessing such memory would be.

Even though D has been desynchronized, there is still a lot of information about it in the abstraction. Specifically, desynchronization saves which function was supposed to have been evaluated, thus it knows not only the state of the program before the function call, but also which function was called. With this information, if the function body was provided later, the analysis could easily resynchronize D with A, B, and C by applying the analysis to that function body starting from D. This is possible because the function symbol was saved along with the region of the heap that it is allowed to affect.

Figure 5.1b shows the result after the call to $\mathrm{fun}_2(\mathrm{B})$. The only accessible region is B and thus it is desynchronized from the A and C regions. Because D is still inaccessible, it just becomes farther in time from being synchronized. However, since the time is not stored as part of the abstraction, and only what is required for resynchronization is stored, it is no more challenging to resynchronize D. Because B and D are completely distinct regions, there is no effect on B (or A or C) when resynchronizing D and thus even though B and D were desynchronized at different times, the resynchronization is no different.

Finally, Figure 5.1c shows the result after the call to $\mathrm{fun}_3(\mathrm{C}, \mathrm{D})$. Because it is possible that the result of region D is accessed here, the same region must be desynchronized again. This results in a nested desynchronization as shown in the dashed box. Both C and D are desynchronized from A, which D is also now desynchronized from C.

To resynchronize everything after Figure 5.1c, the three functions must be evaluated. However, the order in which the functions are evaluated is irrelevant. Evaluating $\mathrm{fun}_1(\mathrm{D})$ first would resynchronize D with C (but not with A). Evaluating $\mathrm{fun}_2(\mathrm{B})$ first would resynchronize B with A. Evaluating $\mathrm{fun}_3(\mathrm{C}, \mathrm{D})$ first would resynchronize C with A and would allow D to be resynchronized with A by only evaluating $\mathrm{fun}_1(\mathrm{D})$.

5.1. Representation of Desynchronization

Definition 4 (Desynchronized Separation). *Desynchronized separation extends a separation-logic-based abstraction, such as HOO, with a desynchronizing term, an extra kind of heap H that represents a desynchronized portion of the heap along with the function to call and the arguments to pass to resynchronize that portion of the heap with the surrounding heap. The heap H is now extended with the following grammar:*

$$\overline{\mathrm{Heap}} \ni H ::= [\![H]\!]\, \mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1, \ldots, \mathrm{V}_n) \mid \ldots$$

To define the concretization of a desynchronized term, concrete values must be extended with functions. These functions do not have any specific semantics, but it is assumed that

while they can mutate the heap, they can only mutate the portion of the heap reachable from global variables, local variables or any closed variables. For unsafe languages such as C this would not be sound, as it is possible to access any object from any other through pointer arithmetic. Essentially, the functions adhere to the standard framing conditions of separation logic [Rey02]. The evaluation of a function is described by the relation

$$\langle\sigma\rangle\textbf{call }v(v_1,\ldots,v_n)\langle\sigma'\rangle$$

which evaluates a call to the function v starting from state σ, passing arguments v_1 to v_n and results in state σ'. Note that all arguments to the function call are fully evaluated before desynchronization, so there is no unnecessary expression evaluation captured here. This minimizes the reachable heap, which, depending on the analysis used to compute the region, may reduce the footprint of the desynchronized term. This is desirable because if less memory is desynchronized, more memory is accessible to the analysis without further desynchronization.

The concretization of HOO with desynchronization is defined as an extension to the concretization of the heap abstraction. Because the signature of the function is not required to change, only the concretization of the new desynchronized terms is provided (where μ may be added as needed):

$$\gamma(\llbracket H\rrbracket\,\textbf{call }\mathrm{V}_f(\mathrm{V}_1,\ldots,\mathrm{V}_n)) \stackrel{\text{def}}{=} \left\{ \eta,\sigma \;\middle|\; \begin{array}{l} (\eta,\sigma_o)\in\gamma(H)\wedge v\in\eta(\mathrm{V}_f) \\ \wedge\,(v_1,\ldots,v_n)\in\eta(\mathrm{V}_1)\times\ldots\times\eta(\mathrm{V}_n) \\ \wedge\,\langle\sigma_o\rangle\textbf{call }v(v_1,\ldots,v_n)\langle\sigma\rangle \end{array} \right\}$$

The concretization concretizes the embedded heap H to a pre-state σ_o and its corresponding valuation. Then for each possible concrete value of the function and each argument, the state σ is the result of evaluating that function on those arguments starting from σ_o. Of course, what makes it possible to reason about applying a function to a portion of the heap is separating conjunction. This dictates that the portion of the heap σ_o was disjoint from the rest of the heap when the desynchronization was created and thus, after this call to a possibly unknown function, σ must be disjoint from the rest of the heap as well.

5.2. Desynchronization with Reachability-based Frame Inference

Desynchronized terms can be introduced at any function call. They are automatically derived by evaluating all of the arguments to symbols, possibly eliminating already existing desynchronized terms to do so. Once this has been completed, a special function reach() is used to determine the desynchronized region.

$$\mathrm{reach} : \wp(\overline{\mathrm{Symbol}})\times\overline{\mathrm{Heap}}\to\overline{\mathrm{Heap}}_u\times\overline{\mathrm{Heap}}_r$$

The function reach() returns a partitioning (H_u,H_r) of the passed heap. The partition H_r is the part possibly reachable from the arguments of the function, including the

global object and any closed variables. The partition H_u is the part unreachable from the arguments of the function. With reach(), a frame H_u is inferred. The introduction of desynchronization is given with a transfer function judgment and then relates a pre abstract state $[H]_{H_0} \mid P$ to a post abstract state $[H']_{H_0} \mid P'$ via a command k:

$$\boxed{\left[[H]_{H_0} \mid P\right] \; k \; \left[[H']_{H_0} \mid P'\right]}$$

Desync-Intro

$$\frac{\begin{array}{c} H = x \mapsto f * y_1 \mapsto \mathrm{V}_1 * \ldots * y_n \mapsto \mathrm{V}_n \\ \mathrm{reach}(\{\mathrm{V}_f, \mathrm{V}_1, \ldots, \mathrm{V}_n\}, H) = (H_u, H_r) \qquad H' = H_u * [\![H_r]\!] \, \textbf{call} \; \mathrm{V}_f(\mathrm{V}_1, \ldots, \mathrm{V}_n) \end{array}}{\left[[H]_{H_0} \mid P\right] \; \textbf{call} \; x(y_1, \ldots, y_n) \; \left[[H']_{H_0} \mid P\right]}$$

If function footprint H_r is over-approximated, i.e. all memory possibly modified by the function call is contained in H_r, the result is fully general. Any client-supplied function can be soundly substituted when resynchronizing. Note that any memory that has been desynchronized is no longer accessible in the analysis, so it may be that, with an imprecise reach(), the analysis cannot proceed. However, it is also possible to not over-approximate the footprint. Doing so means possibly not every function could be substituted for that computation.

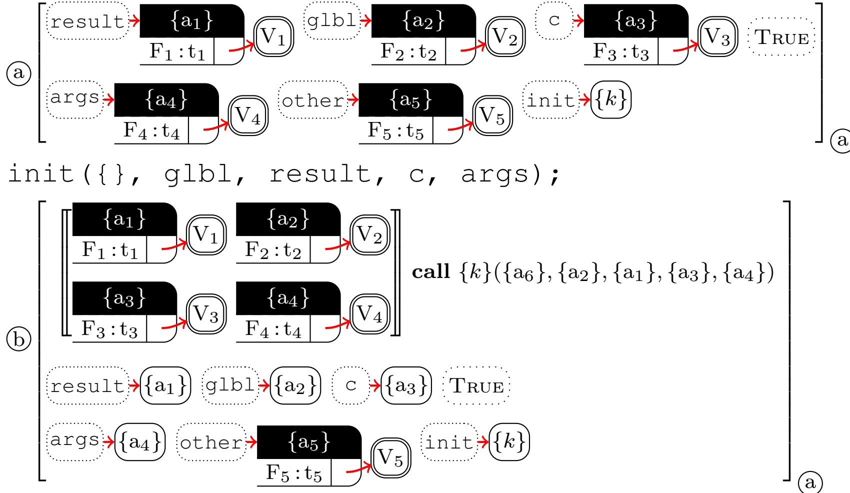


Figure 5.2. – Desynchronized terms are introduced by function calls to unresolvable functions

Example 5.2 (Desynchronization introduction). Figure 5.2 replicates the example showing the call to the client-supplied constructor in the class implementation. This is a function that originated outside the class library and thus is necessarily undefined. When this call occurs, a desynchronized term is introduced to represent the effects of this constructor. The analysis uses an "arrow-following" reach() function that determines that

four objects are reachable in the heap and thus in H_r at ⓐ: $\{a_1\}$, $\{a_2\}$, $\{a_3\}$, and $\{a_4\}$. This leaves all other objects including `result`, `c`, `init`, `args`, `glbl`, `other` $\{a_5\}$, and `other` in H_u. The resulting introduced desynchronized term is as shown in ⓑ.

In other abstract domain operations such as transfer functions, join, widening, or inclusion checking on a domain constructed with desynchronized separation, desynchronized terms must be treated as unknown, but separate portions of the heap. As a consequence, desynchronized memory is inaccessible, as part of transfer functions and any transfer function that must access it may not proceed.

There is a risk that there will be interaction between the library code and the part of the heap accessed by the unknown callback. When this happens, desynchronization is ill-suited for the task. However, in most systems, it is easy (and common) to design programs with encapsulation. When internal library data is fully encapsulated, desynchronization will not include it for client code, which cannot access it.

The inaccessibility of desynchronized terms means that memory that may be modified by a callback cannot be read or written by the other portions of the program. However, by the separation logic frame rule, if a program can be proven without the desynchronized part, it can be proven with a desynchronized part:

$$\textsc{Desync-Frame}\quad \frac{[H \,\vert\, P]\ k\ \left[H' \,\vert\, P\right]}{[H \ast [\![H_d]\!]\,\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1,\ldots,\mathrm{V}_n) \,\vert\, P]\ k\ \left[H' \ast [\![H_d]\!]\,\mathbf{call}\ \mathrm{V}_f(\mathrm{V}_1,\ldots,\mathrm{V}_n) \,\vert\, P\right]}$$

This Desync-Frame rule is a special case of the separation logic frame rule that frames out the desynchronized part of memory and applies the transfer function to the remainder of memory. If this is not well defined because memory in the result of the desynchronized term must be accessed, either a different definition of reach() should be used or the code must be changed to ensure that the needed memory is not in a desynchronized region.

Similarly to the transfer functions, join, widening, and inclusion checking are unable to do much with desynchronized terms. They are treated as syntactic terms and thus join and widening are only allowed to persist only if each desynchronized term is matched with an identical desynchronized term. If they cannot be matched, they must be replaced with a True if the logic supports it. This represents any possible heap soundly. Similarly inclusion checking can only succeed if there is a syntactic match.

Theorem 5 (Soundness of Desynchronization Introduction). *The desynchronization introduction transfer function is sound because*

$$\begin{aligned}&\forall k, \sigma, \sigma', D, D'.\ \langle\sigma\rangle\, k\, \langle\sigma'\rangle\ \textit{and}\ [D]\ k\ \left[D'\right]\ \textit{and}\ \exists\eta.\ (\eta,\sigma) \in \gamma(D)\\ &\qquad \textit{implies}\ \exists\eta'.\ (\eta',\sigma') \in \gamma(D')\end{aligned}$$

5.3. Introduction Heuristics and Resynchronization

For the purposes of analyzing JavaScript libraries, there is a simple introduction heuristic for desynchronized terms: if a function call can be resolved to a known function, a desyn-

chronized term should not be introduced. This policy has the effect that desynchronized terms only represent unknown functions and thus these should never be eliminated from the heap. In fact, they nicely represent the callback behavior that occurs in the library in the library's inferred post-condition.

However, there are circumstances where such a simple heuristic may be non-optimal, and it may be desirable to introduce desynchronized terms even when the code for a called function is available. For example, sufficiently surjective functions [SKK11] are functions where, after a number of recursions, the effect of continued recursion does not matter. In these situations desynchronization can represent the behavior of the unbounded number of recursive calls without actually evaluating all of those calls. Another situation where desynchronization can benefit is in speedup of analysis when known functions may take too long to analyze. If a known function does not affect the heap needed by a piece of code, that function, even though it is known, can be desynchronized to save having to analyze it. In these situations, the post-condition includes a desynchronized term that refers to the known function, but the result of that function has not been evaluated.

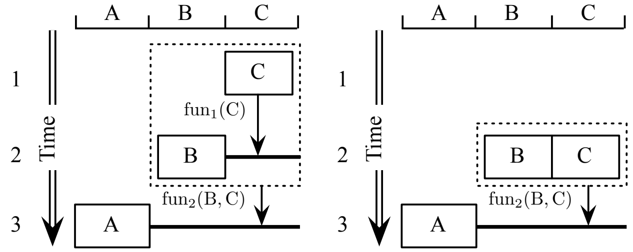
If desynchronized terms are introduced anywhere, it may be necessary for the term describing that memory to be eliminated, due to access of desynchronized memory. This can be done if, for example, the synchronizing function's code is available. The resynchronization process takes advantage of the separation logic frame rule by running the analysis on the synchronizing function starting from the desynchronized term:

$$\textsc{Desync-Elim}\quad \frac{[H_d \mid P]\ \textbf{call}\ \mathrm{V}_f(\mathrm{V}_1,\ldots,\mathrm{V}_n)\ [H'_d \mid P] \qquad [H * H'_d \mid P]\ c\ [H' \mid P]}{[H * [\![H_d \mid P]\!]\,\textbf{call}\ \mathrm{V}_f(\mathrm{V}_1,\ldots,\mathrm{V}_n)]\ c\ [H' \mid P]}$$

With such an elimination rule it is possible to eagerly introduce desynchronized terms on every function call and then lazily eliminate them as portions of the heap are needed.

When employing such an elimination rule, it is possible to consider the variety of ways in which the $[H_d \mid P]$ **call** $\mathrm{V}_f(\ldots)$ $[H'_d \mid P]$ judgment could be satisfied. One way is if each function in V_f can be resolved to known code. In this case the analyzer can be run on each resolvent and a disjunction of post-conditions considered. Alternatively, the formula H could carry the information to satisfy this judgment in the form of a nested Hoare triple [SBRY11].

Example 5.3 (Desynchronization elimination). A region of the heap can be resynchronized by eliminating a desynchronized term:



Here, the region C is resynchronized with B by analyzing the call to $\mathrm{fun}_1(\mathrm{C})$ starting from the memory state C. Note that this resynchronization does not require analyzing $\mathrm{fun}_2(\mathrm{B}, \mathrm{C})$. This combined region can stay desynchronized so long as none of the desynchronized memory is required to proceed with the analysis.

5.4. Related Work

Desynchronized separation is closely tied to the concept of nested Hoare triples [SBRY11] and higher-order separation logic [Kri11]. However, there are several key differences.

The goal of desynchronized separation is fundamentally different from that of nested Hoare triples. Nested Hoare triples are intended to be used in program logics and not for automated reasoning. While there are efforts to automate some amount of reasoning [CHR12], current techniques require significant annotation overhead and perform no inference, only inclusion checking.

The other significant difference is that nested Hoare triples strive for complete generality. A desynchronized term carries the following correspondence with nested Hoare triples:

$$[\![H_1]\!]\ \textbf{call}\ \mathrm{V}_f(\mathrm{V}_1, \ldots \mathrm{V}_n) * H_o \qquad \Leftrightarrow \qquad \exists H_2.\ [H_1]\ \textbf{call}\ \mathrm{V}_1(\mathrm{V}_1, \ldots \mathrm{V}_n)\ [H_2] \wedge H_2 * H_o$$

where H_o is here to illustrate the key differentiating factor. A nested Hoare triple is a pure part of a formula that describes a value, whereas a desynchronized term describes a heap that results from calling a function. The $* H_o$ illustrates which parts of the description are heap and which are pure.

The process of inference using desynchronization is significantly simpler than using nested Hoare triples. This is due to the fact that desynchronization is less expressive than nested Hoare triples. There are fewer existentially quantified variables, and there is no need to treat portions of the heap that are simply passed through the unknown function call as separate portions of the heap that are manipulated. As a result, it is possible to (1) easily adapt existing separation-logic-based analyses to higher-order tasks and (2) easily perform necessary heap splits during the analysis because there are two possible ways the heap can be split.

The key idea of nested Hoare triples is also similar to static contract checking for higher order languages [XJC09, NTHH14], which requires a pure specification of any callback's behavior up front. It is also similar to [MRV12], except that it relies on separation logic and is applied to a stronger heap abstraction.

The goal of desynchronized separation is to not require a specification for callbacks at all, if the developer is judicious with built-in language protection mechanisms. In the event that memory is insufficiently protected, or the reachability analysis is too coarse, desynchronized separation can be trivially extended with nested Hoare triples. In such case, the nested Hoare triple is practically the same as a resolvable function call and no desynchronization is required. However, it is possible to imagine a simpler version of nested Hoare triples where only a footprint for the function is specified. In such a case, desynchronization would be required, but it would be applied only to the supplied footprint.

5.5. Desynchronized Separation Summary

Desynchronized separation extends an existing separation-logic-based analysis with a means for reasoning about calls to unknown functions. This facility supports sound abstraction by incorporating assumptions and the called function symbol into the abstraction itself. This allows any assumptions to be made, including conservative. Thus it can be tuned to support developer-friendly assumptions without significant complexity. As a result, an analysis extended with desynchronized separation is able to precisely abstract a variety of complex function call behaviors.

6. Discussion of Combined Analysis

To analyze JavaScript libraries such as the class library presented in Chapter 3, the set-based HOO abstraction from Chapter 4 combines with desynchronization from Chapter 5. For the analysis of many libraries this is not only a sufficient abstraction, but also a fully precise abstraction, able to infer complete summaries of library behavior. Here, it is targeted at inferring summaries of libraries that extend the base language with new features. These extensions add features such as mixins, traits, classes, and memoization.

This chapter seeks to accomplish four tasks. First, it demonstrates the effectiveness of single-state HOO by comparing its precision with the abstraction used by TAJS, an off-the-shelf abstraction designed for whole programs. Second, it demonstrates the benefits of two-state HOO by showing it can infer summaries of JavaScript libraries. Third, by extending two-state HOO with desynchronization, it shows that libraries that make callbacks, which were previously impossible to analyze with any precision, can now be analyzed. And lastly, it discusses the limitations of the combination of domains.

6.1. Implementing the Combined Analysis

All of the abstractions described in this dissertation are implemented as part of the JSAna static analyzer. JSAna is an analyzer for an extension of the Open-Object-Focused subset of the JavaScript language that is described in Chapter 3. In addition to the formalized subset, it implements other operations supported by the set abstraction such as string concatenation and string constants. This section briefly describes the implementation of the analyzer to give some insight into the architecture.

Figure 6.1 shows an architectural diagram of JSAna. JSAna consists of two main parts: the frontend and the analyzer. The frontend is implemented in JavaScript and runs on a server-side JavaScript engine such as node.js[1]. The frontend converts the textual JavaScript into an intermediate representation more suitable for analysis. It parses the JavaScript with Esprima[2], an ECMAScript 5.1/JavaScript compatible parser that is itself written in JavaScript. Esprima has been extended with support for parsing annotations to indicate preconditions for the analysis. Additionally, some simplification can be performed at this stage, getting rid of extra syntactic structures.

The analyzer part is implemented in OCaml and consists of a rewriter and an abstract interpreter. The rewriter is primarily responsible for converting JavaScript functions into Open-Object-Focused JavaScript functions, including handling binding of functions to objects, converting arguments into an arguments object, and closure conversion. In

[1]http://nodejs.org/
[2]http://esprima.org/

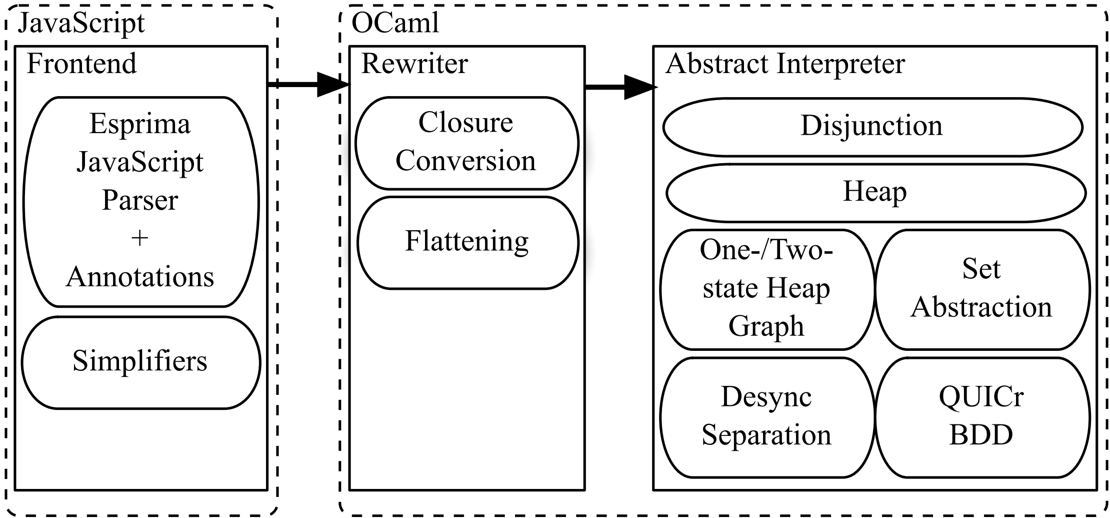


Figure 6.1. – Architecture of the JSAna static analyzer

addition, the rewriter flattens the abstract syntax tree into a-normal form [SF92], so that each statement performs a single action such as looking up an attribute in an object. This process ensures that the abstract interpreter only has to handle the simplest language constructs.

The abstract interpreter is responsible for interpreting the program resulting from the previously described transformations using an abstract domain. The abstract interpreter is implemented as an OCaml functor and an abstract domain is an OCaml module, so the abstract interpreter takes an abstract domain as an argument to produce the actual interpreter. This means that it is easy to swap abstract domains for other domains by simply changing which module is passed to the abstract interpreter.

However, the abstract interpreter is also responsible for managing calls to functions. Function calls can be made to either known functions or unknown functions. If an unknown function is called, desynchronization is triggered.

Many functions in programs, even libraries, however, are known and resolvable. Because JSAna operates on a single function, it is necessary to provide annotations that inform the analysis which functions are bound to certain names. Each known non-nested function can be labeled with an identifier. This identifier can be used as a constant in the constraints representing the precondition. This is able to handle the late binding that JavaScript offers via annotation. If a late binding changes which function is bound to a particular name, the precondition would not be met and the verification would be invalid. The goal of JSAna is to be able to perform verification of libraries that manipulate open objects, not necessarily programs that perform other operations. Specifically, programs and libraries that rely significantly on late binding and functional paradigms are not intended to be supported.

When a function is known and resolved, JSAna performs a fully context-sensitive analysis. It looks up the function in the heap, finds the corresponding code, and runs that

code starting from the heap at the time of the call. It does not perform any generalization or summarization of known functions and consequently does not support recursion. There are many methods for resolving the interprocedural analysis problem [Pnu81, RHS95, RC11], but this problem is not the focus of the dissertation. The fully-context-sensitive approach controls the experiments to ensure that no precision loss occurs due to interprocedural analysis. The annotations provided to the library must supply enough information about the library to resolve any of the library's internal functions.

Resolving functions to determine if they are unknown is a problem that also must be solved. There are three sources of functions:

- Internal functions — Internal functions are functions that are defined within the library function that is being analyzed. That is, they are within the module. These functions are handled precisely by tracking the function symbol and the closure as a symbolic value.
- Library functions — Library functions are functions that are known because they are in the library, but are not defined within the functions that are analyzed. The closures and function symbols must be specified in annotations if calls to these functions should be analyzed.
- Client functions — Client functions exist outside the entire library and are necessarily unknown. They must be handled with desynchronization.

Similarly to the abstract interpreter, the abstract domains are parametric as well. This means that they are also implemented as OCaml functors to ensure that swapping domains is as trivial as possible. For example, the disjunction domain, the heap domain, and the set domain are all parametric abstract domains. This hierarchy of domains forms the final domain that is passed to the abstract interpreter.

Disjunctions are not tracked internally within the heap as described previously. These disjunctions are tracked by a separate disjunction domain at the top level of the hierarchy. In addition to keeping track of the list of disjuncts, the disjunction domain keeps track of the history of how disjunctions were introduced as this gives a good initial guess of how disjunctions should be combined with each other when performing widening. As the number of disjunctions is bounded, when widening, the number of disjunctions is fixed, and then disjuncts will be joined in the heap abstraction, potentially introducing some imprecision.

The heap domain is responsible for managing the heap graph and the set abstraction. As the heap graph is simply the underlying data structure for representing the heap, the heap domain is responsible for manipulating that data structure by adding, removing, and iterating over points-to relationships. The implemented heap graph is a hybrid of the one-/two-state heap graph, where the two states are tracked implicitly as the operations are applied to only the current state throughout the analysis. The desynchronized separation is implemented with tags on the various parts of the heap, where the tags contain the appropriate resynchronization information.

The pure domain, which is responsible for set and value reasoning, is implemented as a slightly specialized version of the QUICr library (described in Chapter 7). This version uses binary decision diagrams [Bry86] to represent QUIC edges and employs only limited use of the base domain inference rules. For applications where there is limited use of constraints on base domain values, this is a significantly faster domain capable of scaling to many more symbols than the fully general QUICr that is described in Chapter 7.

6.2. Single-State HOO Performance/Precision

This section evaluates several hypotheses about single-state HOO: first, that it is fast enough to be useful; second, that it is at least as precise as other open-object abstractions when objects have unknown attributes; and third, that it infers partitions and relations between partitions of unknown attributes precisely enough to verify properties of intricate object-manipulating programs. I used the single-state mode within the JSAna analyzer to analyze a number of small diagnostic benchmarks, each of which consists of one or more loops that manipulate open objects. These benchmarks are drawn from real JavaScript frameworks such as JQuery, Prototype.js, and Traits.js[3]. I chose them to test commonly occurring idioms that manipulate open objects in dynamic languages. To have properties to verify, I developed partial correctness specifications for each of the benchmarks. I then split the post-conditions of the specifications into a number of properties to verify that belong in one of two categories: memory properties assert facts about pointers (e.g., $\mathtt{r} \neq \mathtt{s}$), and object properties assert facts about the structure of objects (e.g., if the object at a_1 has attribute f, then object at a_2 also has attribute f). The full benchmarks and corresponding properties are shown in the Appendix.

I use these benchmarks to compare HOO with TAJS [JMT09], which is currently the most precise (for open objects) JavaScript analyzer. Because TAJS is a whole-program analysis, it is not intended to verify partial correctness specifications and consequently, it adapts a traditional field-sensitive object representation for open objects. However, it employs several features to improve precision when unknown attributes are encountered during analysis: it implements a recency abstraction [BR06] to allow strong updates to objects (but not attributes) on straight-line code, and it implements correlation tracking [SDC+12] to allow statically known attributes to be iteratively copied using **`for-in`** loops.

The results in Table 6.1 show that TAJS and HOO are able to prove the same memory properties. The diagnostic benchmarks are not designed to exercise intricate memory structures, so all properties are provable with an allocation site abstraction. Because both TAJS and HOO use allocation site information, both prove all memory properties.

For object properties, single-state HOO is always at least as precise as TAJS, and significantly more so when unknown attributes are involved. The `static` benchmark is designed to simulate the “best-case scenario” for whole program analyses: it supplies all attributes to objects before iterating over them. Here, TAJS relies on correlation tracking to prove all properties. HOO can also prove all of these properties. It infers a separate

[3] http://jquery.com, http://prototypejs.org, and http://traitsjs.org

Table 6.1. – Analysis results of diagnostic benchmarks. Time compare analysis time excluding JVM startup time. Memory properties compares TAJS and HOO in verifying pointer properties. Object properties compares TAJS and HOO in verifying object structure properties. The # Props columns are the total number of properties of that kind.

	Time (s)		Memory Properties			Object Properties		
Program	TAJS	HOO	TAJS	HOO	# Props	TAJS	HOO	# Props
`static`	0.06	0.09	1	1	1	3	3	3
`copy`	0.13	0.02	1	1	1	0	3	3
`filter`	0.40	0.10	0	0	0	0	6	6
`compose`	0.71	0.54	0	0	0	0	7	7
`merge`	0.19	0.06	2	2	2	0	5	5

partition for each statically known attribute, effectively making it equivalent to TAJS's object abstraction.

The other benchmarks iterate over objects where the attributes are unknown. Here, HOO proves all properties, while TAJS fails to prove any. TAJS's imprecision is unsurprising because correlation tracking does not work with unknown attributes and recency abstraction. However, even if it worked for attributes, recency abstraction does not enable strong updates in loops. HOO, on the other hand, succeeds because it infers partitions of object attributes and relates those partitions to other partitions. In the `copy` benchmark, attributes and values are copied one attribute at a time to a new object. HOO infers that after the iteration is complete, the attributes of both objects are equal. HOO can also verify the `filter` benchmark that requires conditional and partial overwriting of objects. Additionally, HOO continues to be precise, even when complex compositions are involved, as in the `compose` and `merge` benchmarks, which perform parallel and serial composition of objects. For these benchmarks HOO infers relationships between multiple objects and sequentially updates objects through multiple **`for-in`** loops.

While the results indicate that HOO is of comparable performance to TAJS, this is untrue. TAJS has much more complete JavaScript semantics and is thus considering cases that HOO is not. Additionally, HOO's ability to scale is limited by the scalability of the set abstraction, which can be exponential. Here, the performance is good because (1) the benchmarks are small and have relatively few symbols and (2) the exponential cases are not being heavily exercised. This means that the performance is good, even when the constraints are complex. The potentially exponential cost of set abstractions does not mean, however, that HOO cannot scale. Currently, the performance of the set domain is the primary limiting factor because it is not well optimized. This and other limitations will be discussed in Section 6.4.

This evaluation demonstrates that single-state HOO can be effective at representing and verifying properties of open objects, both with statically known attributes and with entirely unknown attributes. Additionally, for cases when objects are fully unknown,

shows that HOO provides significant precision improvement over existing open-object abstractions. Furthermore, it shows that HOO does not need to take a significant amount of time to verify complex properties.

6.3. Two-State HOO with Desynchronization

This section extends the single-state HOO abstraction in two ways. It adds the two-state variant so that full function summaries can be inferred. This includes the addition of attribute/value trackers. Additionally, it includes desynchronized separation building on top of the abstraction provided by HOO. The goal of this section is to evaluate the effectiveness of attribute/value trackers along with desynchronization as extensions to HOO in order to infer precise function summaries for JavaScript libraries that add language features through object manipulation and call backs. This evaluation tests three hypotheses about desynchronization and attribute/value trackers. First, in practice, desynchronization allows summarizing function behavior even when there are callbacks. Second, the reach() function based on heap reachability is sufficient for callbacks made in feature-adding libraries. Finally, attribute/value trackers allow fully precise inference of relations between objects in common object copying and overwriting cases.

To evaluate these hypotheses, I studied a wide variety of JavaScript libraries including JQuery, Prototype.js, Traits.js, MS AJAX, JS.Class, Joose, Classy, Base2, qooxdoo, MooTools, jstraits, light-traits, Closure, and Memoize. I identified several features that are commonly added to JavaScript via libraries: classes, traits, mixins, and memoization. For each of these features, I selected a representative library and extracted one of the core functions of that library, annotating that functionality with pre-conditions. These preconditions indicate aliasing in the heap as well as give names to sets of attributes. Then, on each library, I compared expected postconditions against those generated by the JSAna analyzer.

The summary of results is in Table 6.2. The results are shown across the various libraries and across the properties of the libraries that were checked. For Class, which is similar to the class from the introduction and inspired by a number of the JavaScript class systems, two properties were checked: *resulting object* verifies that the attribute/value pairs are copied from the configuration object without the 'init' attribute precisely; *constructor call* verifies that the constructor calls the user-supplied constructor, with an appropriate portion of the heap such that the result of class construction, which is not returned by the client-supplied constructor, is appropriately constrained in the post-condition. Both of these properties are proven automatically by JSAna.

Two variants of a `memo` function, one of which is synthetic, the other having been extracted with some c changes from the Google Closure library[4], takes a function and returns a memoized version of that function by wrapping it with a function that checks a memoization table first. To extract the code, among other things (fully explained in Appendix E), the use of logical operations was eliminated for simplicity, considering only the case where the local cache is used. What remains is the core functionality of a memo-

[4]https://developers.google.com/closure/library/

ization routine. The first property, *in table*, checks that if a particular set of arguments was memoized, the return value is read from the table and the memoized function is not called. The second property, *call saved*, checks that if a particular set of arguments was not memoized, the return value is computed by calling the argument function and inserting the result into the memoization table. Both properties are automatically proven by JSAna.

There `extend` function from Prototype.js[5], which implements mixins, the synthetic trait composition, and the Traits.js[6] `compose` function copy an object's attributes/values over another object's attributes/values. Only the Traits.js code requires any modification for analysis. It requires changes to avoid the use of arrays and complex boolean operations that are not supported by the analyzer today. The `mixin` function overwrites another object's attributes/values, requiring attribute/value trackers to precisely manage the values and partitions simultaneously. The `compose` function copies two objects onto a new object except that when the attribute names conflict a special conflict value is written to the new object. This requires a more complex partitioning scheme, but exercises attribute/value trackers equivalently. Modulo the additional precision provided by attribute/value trackers, the support for callbacks via desynchronized separation, and the additional cost required from analyzing significantly more complete JavaScript, the results for mixins and traits are comparable to those presented with HOO [CCR14]. The *object extended* property is true if the resulting object has an appropriate partition from each source object and the *conflict managed* property is true if conflicting attributes are correctly assigned their own partition. JSAna automatically proves all of these properties.

Conclusions and understanding the results These results demonstrate that desynchronization can enable analysis of code after a call to an unknown function if the desynchronized region can be determined precisely enough. Of course it is possible to easily construct examples that can defeat the analysis by always passing (possibly useless) memory to the unknown function. However, from manual inspection, it seems that developers do not intend to do this. Typically, they write their code so that there are clean lines of separation, which can be used as part of this discovery process like reach() does.

The results also indicate that attribute/value trackers are critical to the precision of analyses. While HOO is capable of inferring precise partitions of attribute names and values, the addition of attribute/value trackers makes proofs of all of these library properties possible. Only the *in table* and *conflict managed* properties were proven by HOO by itself because these properties reason about a single value and thus strong updates are sufficient to prove them. By using multiple states representing the same trackers from input to output, inferred post-conditions are precisely represented with respect to the trackers used in the pre-condition.

Like single-state HOO, the performance is not tremendous. However, once again, performance problems can be attributed to one component: the set domain. As the

[5]http://prototypejs.org/
[6]http://soft.vub.ac.be/~tvcutsem/traitsjs/

Table 6.2. – Results of running HOO with desynchronized separation and attribute/value trackers on JavaScript meta-feature libraries. Benchmarks above the line are functions constructed to be a simple representative of JavaScript libraries. Below the line are benchmark functions are taken directly from their respective libraries. The following columns indicate which properties were proven with HOO: basic HOO; D: basic HOO + desynchronization; T: basic HOO + attribute/value trackers; D+T: basic HOO + desynchronization + attribute/value trackers. Vars is the peak number of pure symbols used in the analysis. Stmts is the number of statements in the analyzed part of the program (after preprocessing and lowering)

Test	Property	HOO	D	T	D+T	Time (s)	Vars	Stmts
Class (syn)	Resulting Object	✗	✗	✗	✓	8.13	118	128
Class (syn)	Constructor Call	✗	✓	✗	✓	8.13	118	128
Memo (syn)	In table	✓	✓	✓	✓	0.24	179	149
Memo (syn)	Call saved	✗	✓	✗	✓	0.24	179	149
Compose (syn)	Object extended	✗	✗	✓	✓	7.20	111	131
Compose (syn)	Conflict managed	✓	✓	✓	✓	7.20	111	131
`memo`	In table	✓	✓	✓	✓	3.28	357	369
`memo`	Call saved	✗	✓	✗	✓	3.28	357	369
`extend` (mixin)	Object extended	✗	✗	✓	✓	0.16	52	33
`compose`	Object extended	✗	✗	✓	✓	0.98	69	69
`compose`	Conflict managed	✓	✓	✓	✓	0.98	69	69

number of variables grows, the performance of join worsens. The memoization benchmark avoids this problem by not performing any joins. Profiling the slow analyses confirms this by revealing that approximately 95% of the time is spent in the set abstract domain. Discussion about the cause of this slowness can be found in Section 7.6. However, despite this current slowness, because the set abstract domain is easily swappable, it is expected that newer set domains would be easily usable and possibly more performant.

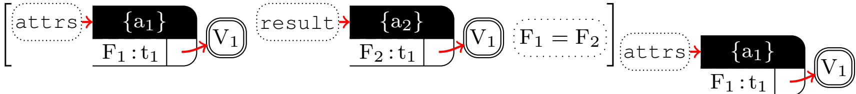
6.3.1. Case Study: Class

The class benchmark is shown in Figure 6.2. It is similar to the function `Class` presented in Chapter 3. The purpose of this section is to analyze the similarities between the actual analysis of this function and the one presented in detail in Chapter 3.

```
function() {
  var copy = function(res,src,exc) {
    for(var p in src) {
      if(!(p in exc))
        res[p] = src[p];
    }
  };
  var result = {}
  copy(result,attrs,{init:""});
  init.apply(result, arguments);
  return result;
};
```

Figure 6.2. – Class instantiation as analyzed

Because this is JavaScript and what was presented in the overview was simplified, there is a great number of extra variables introduced from preprocessing, desugaring, and closure conversion. Modulo these extra variables, the analysis state prior to the call to `init` is exactly as expected given the starting condition. Critically, the `result` and `attrs` objects are the same:



Evaluating the call to `init`, the analysis determines that function that `init` points to is unknown. This triggers a heap traversal to evaluate reach(). Because `attrs` is not reachable and because of the call to `copy`, `attrs` and the object it points to are appropriately protected from the reachability query. As a result, essentially the only memory remaining outside the desynchronized region is local variable declarations and the object pointed to by `attrs`. This corresponds nicely with the Chapter 3 example.

Throughout the analysis from the beginning of the program to the call of `init`, objects are manipulated materializing individual attributes, performing selective copies and

summarizing attributes. During materialization trackers are appropriately replicated across both new partitions of the object. When the copy is performed, the attribute is transferred from one object to another. Due to the fact that the target object ($\{a_2\}$) materializes the attribute that is being written and removes the old attribute in the process, the attribute/value tracker is safely transferred. As a result, by the time the analysis reaches the call to `init` it has a fully precise view of the heap up to the point of desynchronization.

6.3.2. Case Study: Memoization

```
function() {
  var key = uid(arguments);
  if(!(key in memo))
    memo[key] = f.apply(null,arguments);
  return memo[key];
}
```

Figure 6.3. – Inner code for memoization function

The memo benchmark (Figure 6.3) transforms a function into a memoized version of that function. To accomplish this, it first translates the arguments array into a unique identifier by calling a `uid()` function passing it the entire arguments object. Then it determines if that unique identifier is already in the memoization table. If so, it returns the value from the table. Otherwise, it calls the function to be memoized f passing it arguments (via JavaScript's apply functionality) and then memoizes the result.

Each of the function calls is challenging. The `uid()` function is essentially a hash function. It is responsible for converting data of any type into a unique string suitable for use in indexing into an object. Because hash functions are typically hard to analyze and this is a hash function that hashes to strings, this function presents a problem for analysis. Even if we had the code for it, it would be undesirable to analyze it.

The second function call is also challenging because it is a callback into client-supplied code. The behavior of the function could be anything. It could have side effects or it could be pure. Its only restriction is from JavaScript being memory safe (it cannot create pointers to previously unreachable parts of the heap).

Both of these problems are addressed by desynchronization as shown in Figure 6.4. This figure shows the representation of the post-condition of the function returned by calling the memo() function. In it we can see that not only was the callback to the client-supplied function f() desynchronized, but also the call to `uid()` was desynchronized. Additionally, because the arguments object may have been modified by the `uid()` function, it is necessary to nest the desynchronizations to represent the result.

Nested desynchronization allows continuation-like behavior to be analyzed over parts of the program. Here the arguments object was possibly modified by the `uid()` function before being possibly modified by the callback. The benefit of this nested structure is

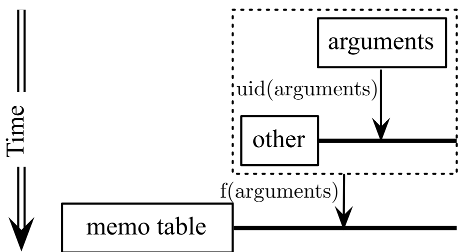


Figure 6.4. – Desynchronization phases of the memoization example

even if there is a sequence of functions that all touch the same memory, analysis can proceed by nesting all of these individual functions.

6.4. Boundaries of Analysis and Future Improvements

Precision Limitations While the results suggest that all of single-/two-state HOO along with desynchronization and attribute/value trackers can be effective on JavaScript code, there are limitations to the precision. Without attribute/value trackers, HOO is unable to keep precise track of which values in some value set V each attribute in some attribute set F points to. Attribute trackers remedy this situation, but only in certain circumstances. For example, complex, nested copies are not currently supported by these trackers; the following code wraps each value inside a newly allocated object.

```
result[a] = {value: attrs[a]};
```

Without the ability to reason about intermediary objects, full precision cannot be maintained and such abstractions fall back to what HOO can do. However, this behavior does not appear to occur in most libraries and thus may not be a significant issue. Adding support for this particular case is simply another form of tracker, but the inference of such trackers remains challenging.

Another solution to this problem would be to require a richer value domain that is not simply sets, but one that supports maps. The problem here is that maps are difficult to precisely abstract. HOO is, in essence, an abstraction of maps that sometimes loses specific key/value relationships by abstracting a collection of keys as a set and a collection of corresponding values as a set. Requiring a value domain accomplishes nothing because it is currently unknown how to construct such a value domain without techniques such as those already employed by HOO.

Another cause for concern is the capabilities of desynchronization. It was intentionally chosen to be simple so as to be easy to infer. However, there are a number of possible extensions to consider.

One is to allow read-only portions of memory to be accessed by a function. Right now desynchronization does not distinguish between read-only and read-write memory, that is memory that may only be read by the unknown function versus memory that may be

both read and written by the unknown function. Adding such functionality is likely not difficult, but it significantly complicates the reach() function.

Another possible extension is to enable accessing desynchronized memory after desynchronization. Because JavaScript is a safe language, this is always possible, but many commands might raise exceptions, which could cause a massive explosion in (probably spurious) program paths after desynchronization. It is possible to make assumptions about those paths, once again saving those assumptions into the desynchronization, but it is unclear at this point how many tangible benefits this will provide.

While the current reach() implementation appears to be suitable for the analysis of many libraries, it may be overly pessimistic. In certain situations developers intentionally make portions of libraries globally mutable, but mutation is still not the common case. Consequently, it may be desirable to develop a reach() that makes use of annotations provided in the precondition to restrict certain parts of memory to not be desynchronized. Doing so will improve the precision of the analysis and cause fewer nested desynchronizations to occur as a result of inaccessible memory.

Performance Limitations In its current state, HOO does not scale. With small inner loops, HOO can be very fast, as shown in Table 6.1. Unfortunately, as the number of symbols in the heap climbs, HOO's join performance suffers significantly as Table 6.2 shows. This is due to two problems that revolve around the set domain: the join makes too many queries to the set domain and the set domain is too slow at both handling queries and performing domain operations.

The use of the matching algorithm in the join requires the use of many set queries. Because, in nearly all cases, the implementation resorts to using the allocation site information, it is entirely unnecessary to perform all of these queries. If the allocation site information were not so necessary for the join algorithm, the join algorithm would benefit from all of these queries, but as is it today, these queries are largely unnecessary. It is likely that removing this would improve performance. However, most of the performance cost is in performing the join in the set domain.

One other problem with scalability is how the set domain is utilized. HOO introduces a set symbol that must be handled by the set domain for nearly everything. The number of these symbols is as follows:

$$\#\text{syms} \geq \#\text{vars} + \#\text{objs} + \#\text{attrs} + \#\text{vals}$$

where #vars is the number of program variables active in the program, #objs is the number of abstract objects (at least the number of allocations sites), #attrs is the total number of partitions for all abstract objects, and #vals is the total number of abstract values in the program. If the set abstraction cannot cope with a large number of symbols this use is prohibitively costly.

JavaScript Feature Limitations The analysis today is designed to work on a core language of JavaScript. This language does not support all of JavaScript's features. For example, the language does not support numbers, Booleans, dates, and arrays. As

a result it also does not support many operations such as type coercions, `eval`, and implicit prototype behavior. Adding support for this functionality to the analysis varies in difficulty. Adding support for new values requires replacing the set domain with a domain that represents sets of values of the appropriate types (such as strings, numbers, and Booleans simultaneously).

Improving support for prototypes could significantly enhance support for fully modular JavaScript analysis. The challenge is summarizing a function for any prototype chain. If the prototype chain is unimportant to the function (because all accessed attributes necessarily exist within the first objects of prototype chains), the HOO abstraction already supports this. However, more complicated situations can arise that require extensions beyond the separation logic used here [GMS12]. However, the automation of such extensions remains unclear.

Support for arrays is mostly a question of adding all of the array manipulating library routines. HOO's representation of objects is sufficient for also representing arrays. However, it might be advantageous to consider adapting a specialized array analysis [CCL11] for the job. Such a specialized array analysis is designed to take advantage of the fact that arrays indexes are typically consecutive to aid the abstraction.

7. Set Abstraction: Relational, First-Class Sets

A significant part of the HOO abstraction is the use of a set abstraction to represent relationships between sets of addresses, attributes, and values. In previous chapters, it was assumed that a precise representation of sets was provided as a parameter. However, this abstraction for sets was not detailed. This chapter is an extended version of [CCS13], which presents an analysis that abstracts containers by the set of elements contained in them to infer facts about (a) the possible set of values in a container; and (b) how these values relate to values stored in other containers. In general, one may envision two main types of static analyses: (1) *content-centric* analyses that infer assertions for the possible sets of values in each container, *in isolation*; or (2) analyses that infer relations between the values stored in various containers, *as-a-whole*.

```
def extendClass(F1):
  F2 = set([f for f in F1
        if f >= c])
  return F2
```

Figure 7.1. – Sample program that filters a set F_1, producing a set F_2

To illustrate this difference, consider the Python code function `extendClass` in Figure 7.1 (the name of this function will become clearer below). This function takes a set F_1 and returns a set F_2 where F_2 is the subset of elements from F_1 such that each element is greater than or equal to some variable `c`. An important post-condition of `extendClass` is $F_2 \subseteq \{ \nu \in F_1 \mid \nu \geq c \}$, but neither the content-centric nor the as-a-whole analyses can produce this post-condition. The content-centric analysis, which represents each set F_1 and F_2 as individual variables in a domain for reasoning about values, would produce $F_2 \subseteq \{ \nu \mid \nu \geq c \}$ where ν ranges over the universe of values. Because all values of F_1 are not related to all values of F_2 in some way, a content-centric analysis cannot represent any relationship between F_1 and F_2. As-a-whole analyses, which reason only about the relationships between sets, can produce $F_2 \subseteq F_1$, but fail to infer anything about the individual elements of F_2. By combining these two classes of analyses, our analysis finds the desired invariant: $F_2 \subseteq \{ \nu \in F_1 \mid \nu \geq c \}$.

The `extendClass` function is abstracted from a function in Processing.js[1]. A simplified version of the original function is shown in Figure 7.2 (in Python for consistency with the evaluation); the original set version models the key set of this dictionary version.

[1] http://processingjs.org/

```
def extendClass(D):
  E = {k:v for k,v in a.iteritems()
         if k >= "%"}
  return E
```

Figure 7.2. – `extendClass` function extracted from Processing.js

This function copies a dictionary containing a number of values to another dictionary. It only copies those elements that start with letters higher than % in the ASCII table, specifically excluding keys starting with $. These dictionaries are used as objects, and in the context of this framework, $ is interpreted as private and thus should not be copied. Functions like this one are pervasive in programs written in dynamic languages because most run-time structures are implemented using dictionaries (or objects, maps, or tables) and those run-time structures are directly accessible by the developer and can be modified. As a result, previously simple operations such as inheriting a class become complex dictionary manipulations involving copy operations. To statically analyze programs written in dynamic languages, this chapter presents powerful new static analysis techniques that can reason about these kinds of functions.

This analysis tracks (subset) inclusion relations between expressions involving set abstractions of containers through a special graph structure called a QUIC graph. A QUIC graph is a succinct encoding of *set expressions* and *inclusion relations* between them. The expressions represented by a QUIC graph are (1) basic, *atomic sets* that abstract the set of values stored in a container and singletons created by scalar expressions; (2) restricted unions and intersections of the atomic sets; and (3) comprehensions of set expressions through first-order predicates. The predicates are captured by an arbitrary base domain, which can reason about program variables and formal bound variables that represent the scalar-valued contents of a basic set. The QUIC graph is thus a compact structure for storing a conjunction of subset constraints between set expressions. This chapter defines QUIC graphs and builds abstract domain operations over these graphs. The QUIC graph domain is designed to yield a tight integration between the base domain and the QUIC graph domain so that the resulting analysis can transfer facts from one domain to another, quite seamlessly.

The content-centric analysis of containers is rather well understood (e.g., [GRS05, CCL11, DDA11]). Such analyses focus on strategies for partitioning or splitting *summary variables* that smash the contents of the container into an essentially weakly-updated scalar variable. These techniques are orthogonal and complementary to QUIC graphs. With summary variables, one might capture independent comprehensions, such as $\mathrm{F}_1 \subseteq \{ \nu \mid p(\nu) \} \wedge \mathrm{F}_2 \subseteq \{ \nu \mid p'(\nu) \}$ for some predicates p and p'. If the predicates p and p' are the same or related, then these facts may indirectly imply a relation between F_1 and F_2 but essentially only through their contents. On the flip side, the pure container-as-a-whole approach would track relations directly between F_1 and F_2 without characterizing their contents. Some existing containers-as-a-whole approaches incorporate some fixed content reasoning (e.g., [PTTC11]). This chapter presents a tight integration of these

two approaches with domains for reasoning about scalar variables and their relations to the set elements. As a result, the QUIC graph domain promises to be a lot more powerful than a simple conjunction of both individual domains.

The QUIC graph domain is implemented for a simple imperative programming language with integers and sets (of integers) in addition to being integrated with the HOO domain presented in prior chapters. This language captures basic arithmetic over integers and operations over sets such as union, intersections, difference, insertion/deletion of elements, and iteration over sets. The resulting analyzers use the QUIC graph domain, as well as two domains representing the content-centric and container-as-a-whole approaches. The evaluation was carried out by translating a variety of set manipulating programs from the Python test suite. The results are quite promising: the QUIC graph domain is more precise than the other domains, proving more properties than a simple combination of a content-centric approach and a container-as-a-whole approach.

This chapter describes the following aspects of QUIC graphs:

- It identifies the need for simultaneous reasoning about containers as-a-whole *and* their contents to enable modular, precise reasoning of container-manipulating programs (Section 7.1).
- It describes QUIC graphs to represent universally-Quantifed Union and Intersection set Constraints in a canonical manner using a hypergraph data structure. It builds an abstract domain (functor) based on QUIC graphs. A novel aspect of the domain is the use of predicate edge labels to capture set comprehensions (Section 7.2).
- It presents a framework for *inference* using QUIC graphs. It shows how to utilize the structure of QUIC graphs to compute all logical implications of a given QUIC graph. It presents the inference procedure for strengthening base domain invariants within a QUIC graph. Finally, it shows how laziness significantly improves the cost of inferring consequences of QUIC graphs, and describe an efficient implementation (Section 7.3).
- It defines an abstract domain using QUIC graphs with inference and show how all domain operations and reductions are easily implemented using lazy inference (Section 7.4).
- It evaluates the effectiveness of the abstract domain on a set of benchmarks from the Python test suite. For a reasonable performance overhead, the QUIC graphs abstract domain is significantly more precise than either a content-centric or a container-as-a-whole approach and unlike the content-centric and container-as-a-whole approaches can automatically prove most properties specified in the Python test suite for set operations (Section 7.5).

7.1. Overview

This section walks through inferring the desired post-condition for the `extendClass` example to highlight the main challenges in obtaining precise combined content-as-a-whole

$$
\begin{array}{rcl}
\text{program} & ::= & \text{decl}^{*}\ \text{stmt}^{*} \\
\text{decl} & ::= & \textbf{int}\ \text{scalarVar} \mid \textbf{set}\ \text{setVar} \\
\text{stmt} & ::= & \text{scalarVar}\ \mathbf{:=}\ \text{scalarExpr} \\
 & \mid & \text{setVar}\ \mathbf{:=}\ \text{setExpr} \\
 & \mid & \textbf{loop}\ \text{stmt}^{*} \\
 & \mid & \textbf{branch}\ \text{stmt}^{*}\ \textbf{orelse}\ \text{stmt}^{*} \\
 & \mid & \textbf{havoc}\ \text{setVar} \\
 & \mid & \textbf{assume}\ \text{conditional} \mid \textbf{assert}\ \text{conditional} \\
\text{setExpr} & ::= & \emptyset \mid \{\text{scalarExpr}\} \mid \text{setVar} \mid \text{setExpr} \cup \text{setExpr} \\
 & \mid & \text{setExpr} \cap \text{setExpr} \mid \text{setExpr} \setminus \text{setExpr} \\
\text{scalarExpr} & ::= & \text{scalarVar} \mid \text{scalarConst} \mid \text{scalarUnary}(\text{scalarExpr}) \\
 & \mid & \text{scalarBinary}(\text{scalarExpr}, \text{scalarExpr}) \mid \textbf{choose}(\text{setExpr}) \\
\text{conditional} & ::= & \text{scalarConditionals} \mid \text{setExpr} \subseteq \text{setExpr} \mid \text{scalarVar}\ \textbf{in}\ \text{setVar} \\
\text{setVar} & ::= & \mathrm{F} \\
\text{scalarVar} & ::= & \mathrm{f} \\
\text{scalarConst} & ::= & \mathtt{c}
\end{array}
$$

Figure 7.3. – An imperative, set-manipulating programming language. A sequence of a symbol α is written as α^*.

invariants that motivate our design of the QUIC graph domain. At a high-level, deriving the desired post-condition for the `extendClass` function requires the careful application of transitive closure of inclusion constraints, an effective reduction [CC79] strategy with base domain elements, and a non-trivial join operator.

7.1.1. Set Language

For the purposes of evaluating sets, this chapter uses an imperative programming language with scalar values and set values whose elements are scalars, shown in Figure 7.3. scalar operations (e.g., addition, subtraction, multiplication, and division) are assumed to be given as unary or binary operators (scalarUnary or scalarBinary, respectively). For convenience, a single scalar type (integers) is fixed in the language. Unless otherwise mentioned, sets are assumed to range over this type (integers). However, the framework is quite general. Because it only assumes the base domain is a sound abstract domain, it can handle a variety of types including integers, floats, and strings by using base domains designed to reason over scalar variables of those types. This chapter does not address sets of sets or complex structures such as lists. However, the framework can be extended to handle these types by instantiating with more complex base domains such as another domain for sets.

For the purposes of analysis, an input program is lowered introducing additional instrumentation variables. The lowering converts all loops (e.g., **for-in**) into a single non-deterministic **loop** construct and all conditional statements into a non-deterministic **branch** construct. The **havoc** statement is an arbitrary value assignment for modeling unknown effects, and the **assume** statement is used to encode the conditions in each

```
def extendClass(F₁) {
  F₂ := ∅;
  for (f in F₁) {
    if (f > c) {
      F₂ := F₂ ∪ {f};
    }
  }
  return F₂;
}
```

```
def extendClass(X) {
  Fₒ := ∅; Fᵢ := F₁; F₂ := ∅;
  loop {
    assume Fᵢ ≠ ∅;
    f := choose(Fᵢ); Fᵢ := Fᵢ \ {f};
    branch {
      assume f > c; F₂' := F₂ ∪ {f};
      F₂ := F₂';
    }
    orelse {
    }
    Fₒ := Fₒ ∪ {f};
  }
  assume Fᵢ = ∅;
  return F₂;
}
```

Figure 7.4. – Left: the `extendClass` example that filters positive elements from a set F_1 into a set F_2. Right: its lowered version.

branch. One key instrumentation transforms each **for-in** loop over a set F_1 to introduce two sets F_o, F_i that are assumed to partition F_1 (i.e., $F_1 = F_i \uplus F_o$). The set F_o represents all variables that have been iterated over thus far. Likewise, F_i represents the elements of F_1 that remain to be iterated over. The iteration order is assumed to be non-deterministic. The loop exits when $F_i = \emptyset$ or alternatively $F_o = F_1$. It is assumed that iterations over a set F_1 do not modify F_1 in the body of the loop (as is the standard semantics for container iteration).

Example 7.1. Figure 7.4 (left) shows a translation of the Python `extendClass` example from Section 7 to an imperative, set-manipulating program. This program filters elements from an input set F_1 greater than or equal to c into a set F_2. The set F_2 is a variable introduced in the translation to name the set being constructed by the comprehension. The lowered version of this program is also shown alongside (right).

7.1.2. Motivating Example

Figure 7.5 shows an annotated lowered version of the `extendObject` from Figure 7.1. In first three commands, set F_i is initialized to F_1, while F_o and F_2 are initialized to the empty set $\emptyset$. The `extendObject` loop begins before point ①. An arbitrary element f is chosen out of set F_1 after point ① with the **choose** statement and removed from set F_i. The element f is added to set F_2 in the first case of the non-deterministic **branch**, while

```
F_o = ∅;
F_i = F_1;
F_2 := ∅;
loop {
①[F_1 = F_i ∪ F_o ∧ F_2 ⊆ { ν ∈ F_o | ν > c }]
  assume F_i ≠ ∅;
  f = choose(F_i);
  F_i = F_i \ {f};
  branch {
    assume f > c;
    F'_2 = F_2 ∪ {f};
②[F_1 = F_i ∪ {f} ∪ F_o ∧ F_2 ⊆ { ν ∈ F_o | ν > c }
  ∧ F'_2 = F_2 ∪ {f}                                   f > c]
③[F_1 = F_i ∪ {f} ∪ F_o ∧ F_2 ⊆ { ν ∈ F_o | ν > c }
  ∧ F'_2 = F_2 ∪ {f} ∧ {f} = { ν ∈ {f} | ν > c }       f > c]
    F_2 = F'_2;
④[F_1 = F_i ∪ {f} ∪ F_o ∧ F_2 ⊆ { ν ∈ F_o ∪ {f} | ν > c }   f > c]
  }
  orelse {
    assume f <= c;
⑤[F_1 = F_i ∪ {f} ∪ F_o ∧ F_2 ⊆ { ν ∈ F_o | ν > c }   f ≤ c]
  }
  F_o := F_o ∪ {f};
}
assume F_i = ∅;
⑥[F_2 ⊆ { ν ∈ F_1 | ν > c }]
```

Figure 7.5. – Inferring QUIC graph invariants on the `extendClass` example.

set F_2 is left unchanged in the other. The final assignment in the loop simply moves the element f into set F_o to continue the iteration.

The boxed formulas in Figure 7.5 are program invariants that are inferred (under the pre-condition TRUE), that is, the fixed-point result of an abstract interpretation. Our goal is to be able to derive the post-condition $F_2 \subseteq \{ \nu \in F_1 \mid \nu > c \}$, that is, output set F_2 is a subset of the positive elements of the input set F_1, at program point ⑥. Here and in the rest of this chapter, ν is used as the bound variable for all comprehensions. This figure selectively shows the key constraints needed to derive this post-condition. The first observation is that although inclusion constraints plus comprehension expressions are sufficient to state the desired post-condition, the inferred loop invariant at point ① requires a more expressive set expression language (i.e., union expressions). It is straightforward to see that this loop invariant $F_1 = F_i \cup F_o$ along with the loop exit condition $F_i = \emptyset$ implies the desired post-condition and that the initial state where $F_i = F_1 \wedge F_o = F_2 = \emptyset$ implies the loop invariant.

Consider the fixed point iteration of the loop (i.e., showing that loop invariant is inductive and thus consecutes) and focus on the transition to invariant ②—the difference with respect to the loop invariant is shown shaded. This transition begins with the addition of element f to set F_2. The **`assume`** is reflected in the invariant with a base domain constraint $f > c$ shown to the right in the box. It is necessary to transfer the relationship between F_2 and F_o to F_2' and F_o to generate the desired function post-condition. Knowing when to transfer these relationships by transitivity is critical to both performance and precision. The QUIC graph representation allows us to limit the guesswork of when to apply the various transitivity rules to derive additional facts.

Invariant ③ shows a reduction step that transfers information from the base domain to the QUIC graph domain. In particular $f > c$, so it is also the case that $\forall \nu \in \{ f \} . \nu > c$ (i.e., applying a $\forall$-introduction rule). In terms of QUIC graphs, any constraint of the form $\{ f \} \subseteq \{ \nu \in \bar{T} \mid B[\nu] \}$ can be strengthened to $\{ f \} \subseteq \{ \nu \in \bar{T} \mid B[\nu] \wedge \nu > c \}$ where B is a predicate described by the base domain and $\bar{T}$ is any basic set expression, including $\{ f \}$. For abstract interpretation, the conjunction $\wedge$ becomes a meet operator $\sqcap$ on base domain elements. Thus, $\{ f \} \subseteq \{ \nu \in \{ f \} \mid \nu > c \}$ is shown in invariant ③. This "seed" constraint is sufficient to derive other ones, such as $\{ f \} \subseteq \{ \nu \in F_2' \mid \nu > c \}$, by transitivity on demand. The QUIC graph structure with singleton known-scalar sets enables an eager transfer of information from the base domain coupled with lazy propagation of this information (see Section 7.3).

This reduction step is used for deriving the invariant at point ④. Point ④ shows the invariant derived from ③ by projecting out F_2 (and then renaming F_2' to F_2). From invariant ③, intuitively $F_2' \subseteq \{ \nu \in F_o \mid \nu > c \} \cup \{ \nu \in \{ f \} \mid \nu > c \}$ is true by applying transitivity (and that union with any set is monotonic), so that $F_2' \subseteq \{ \nu \in F_o \cup \{ f \} \mid \nu > c \}$, which gets to the desired result after projecting the old F_2 and renaming F_2' to F_2. It is not difficult to check this step; rather, the main challenge in an automated analysis is guessing that these are the appropriate steps to obtain the desired invariant. For example, both $F_2' \subseteq \{ \nu \in F_1 \cup \{ f \} \mid \nu > c \}$ and $F_2' \subseteq F_o \cup \{ f \}$ are sound over-approximations of the projection that are syntactically close, but nowtoo much precision has been lost to get the desired post-condition. From the QUIC graph perspective, this derivation is

a propagation of facts across nodes and edges that can be done on demand by the *lazy closure* (see Section 7.3).

The invariant at point ⑤ in the unchanged case entails the invariant that was just computed at point ④ (except for the base domain constraint), so the result of the join at program point 21 is the invariant at point ④ without the base domain constraint f > c, and after the assignment, the result is exactly the loop invariant at point ①.

In summary, it is difficult to derive enough constraints via transitivity and strong enough ones via reduction from the base domain. On the flip side, transitive closure, even with restricted union and intersection constraints, is exponential (see Section 7.3). The QUIC graph representation eases this tension by representing inclusion constraints over unions, intersections, and comprehensions in a canonical manner that facilitates on-demand propagation of information.

7.2. QUIC Graphs

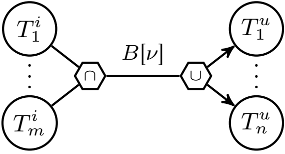
A *Quantified Union/Intersection Constraint graph* is a graph data structure that represents inclusions between set expressions. Throughout the rest of the chapter the notation F is used with subscripts to represent set variables and f with subscripts to represent base domain variables. The special variable ν will be used as a formal bound variable for set comprehensions, as will be explained in this section. The symbol T represents *atomic set expressions* – one of three possible elements: the empty set $\emptyset$, a singleton set containing a base domain variable $\{\,\mathrm{f}\,\}$ or a set variable F_1. The symbols $\bar{T}^i, \bar{T}^u$ represent a number of Ts in an intersection or a union respectively.

Definition 5 (QUIC edge). *Let $T_1^i, \ldots, T_m^i = \bar{T}^i$ and $T_1^u, \ldots, T_n^u = \bar{T}^u$ be symbols representing finite sets and B be a base domain abstract state involving a bound variable ν, acting as a predicate where $\top$ is* true *and $\bot$ is* false*. A* QUIC edge *is a constraint*

$$\bigcap_{i=1}^{m} T_i^i \subseteq \left\{ \nu \in \bigcup_{j=1}^{m} T_j^u \,\middle|\, B[\nu] \right\} \text{ represented using the notation } \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_B$$

which is an edge in an edge labeled hypergraph.

Dots above the set operators make clear that set operators are part of the syntax of a QUIC constraint or a QUIC edge. Graphically a QUIC edge is represented as a hyperedge:



For convenience, if there is only one T in the union (respectively intersection), the union (respectively intersection) is elided from the figure. Additionally, if the label $B[\nu]$ is top in the base domain, the label is elided from the edge.

Definition 6 (QUIC graph). *A QUIC graph $G \in \widetilde{G}$ is an edge labeled hypergraph constructed of QUIC edges. It represents a conjunction of constraints where each constraint corresponds to one QUIC edge in the graph. A QUIC graph has the following syntax:*

$$\begin{aligned} G ::= \;& G_1 \;\wedge\; G_2 \\ \mid \;& \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_B \end{aligned}$$

A QUIC graph is a canonical representation of the set of conjoined constraints. It is designed to be compact and to allow efficient inference operations (see Section 7.3).

Following is a series of examples to demonstrate the QUIC graph representation.

Example 7.2 (Basic QUIC graphs). Consider that would be produced after line 1 from the example in Figure 7.5:

$$\mathrm{F_o} \subseteq \emptyset \;\wedge\; \mathrm{F_i} \subseteq \mathrm{F_1} \;\wedge\; \mathrm{F_1} \subseteq \mathrm{F_i} \;\wedge\; \mathrm{F_2} \subseteq \emptyset$$

This is represented as a QUIC graph:

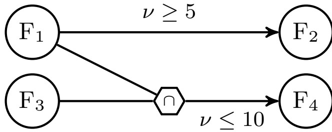
Unlike the constraint formula, the symbols $\mathrm{F_1}$, $\mathrm{F_i}$, and $\emptyset$ only occur once in the graph. This makes the relationships more clear and eliminates possible redundancy.

Conjoining multiple constraints produces a QUIC graph with multiple edges and including unions or intersections requires a hypergraph to show the relationships:

Example 7.3. To encode the formula:

$$\dot{\bigcap} \mathrm{F_1} \dot{\subseteq} \dot{\bigcup} \mathrm{F_2} \Big|_{\nu \geq 5} \wedge \dot{\bigcap} \mathrm{F_1}, \mathrm{F_3} \dot{\subseteq} \dot{\bigcup} \mathrm{F_4} \Big|_{\nu \leq 10}.$$

The following hypergraph is used:



To be practical, a representation for set constraints cannot stand alone. There must be a way to represent relationships between sets and base domain variables as well. To do this one constructs a combined domain where elements are pairs $(G, B) \in \widetilde{S} = \widetilde{G} \times \widetilde{B}$ where G is a QUIC graph domain instance and B is a base domain instance. Note that the base domain has two roles: (a) it labels edges in the QUIC graph and (b) it captures invariants on base domain variables.

To specify the concretization for both QUIC graphs and QUIC graphs combined with an external base domain, a concretization (where γ is overloaded for all concretizations) for the base domain is required:

$$\gamma : \widetilde{B} \rightarrow \wp((\textsc{BaseVar} \rightarrow \textsc{BaseVal}) \times \wp(\textsc{BaseVal}))$$

The symbol BaseVar is all base domain variables, BaseVal is all base domain values. This is a non-standard concretization because given some abstraction, it returns a set of functions that map base domain variables to base domain values and for each function, there is a corresponding set that contains the base domain values to which the bound variable ν can be assigned. This is used to define concretization for QUIC graphs

Definition 7 (Concretization). *The concretization γ of a QUIC graph G has the following type, given that* SetVar *is all set domain variables:*

$$\gamma : \widetilde{G} \rightarrow \wp((\textsc{SetVar} \rightarrow \wp(\textsc{BaseVal})) \times (\textsc{BaseVar} \rightarrow \textsc{BaseVal}))$$

Where the result is a set of pairs of functions (η, η_B), where η maps set variables to sets of base domain values and η_B maps base domain variables to base domain values. These two functions mappings are valid with respect to constraints both on the sets and on the base domain.

The concretization function is then defined as such:

$$\gamma(G_1 \wedge G_2) \stackrel{def}{=} \{ (\eta, \eta_B) \mid (\eta, \eta_B) \in \gamma(G_1) \text{ and } (\eta, \eta_B) \in \gamma(G_2) \}$$

$$\gamma\left(\dot{\bigcap} [T_1^i, \cdots, T_n^i] \dot{\subseteq} \dot{\bigcup} [T_1^u, \cdots, T_m^u] \Big|_B \right) \stackrel{def}{=}$$

$$\left\{ (\eta, \eta_B) \;\middle|\; \begin{array}{l} (\eta_B, \bar{b}) \in \gamma(B) \text{ and} \\ \text{for all } \nu.\ \big(\nu \in \eta(T_1^i) \text{ and } \cdots \text{and } \nu \in \eta(T_n^i)\big) \\ \qquad \text{implies } \big(\nu \in \bar{b} \text{ and } (\nu \in \eta(T_1^u) \text{ or } \cdots \text{or } \nu \in \eta(T_m^u))\big) \end{array} \right\}$$

The concretization for a combined domain S is the same set of pairs (η, η_B), so the type and concretization follow:

$$\gamma : \widetilde{G} \times \widetilde{B} \rightarrow \wp((\textsc{SetVar} \rightarrow \wp(\textsc{BaseVal})) \times (\textsc{BaseVar} \rightarrow \textsc{BaseVal}))$$

$$\gamma((G, B)) \stackrel{def}{=} \{ (\eta, \eta_B) \mid (\eta, \eta_B) \in \gamma(G) \text{ and } (\eta_B, \bar{b}) \in \gamma(B) \}$$

Expressivity: Now, the expressivity limitations of QUIC graphs are discussed. As such, QUIC graphs allow unions, intersections and comprehensions of sets but in a restricted manner. The design choices are motivated with the following.

The first expressivity restriction arises from the manner in which comprehension is introduced in our language. For instance, QUIC graphs are able to express inclusions of the form $\mathrm{F}_1 \subseteq \{\nu \in \mathrm{F}_2 \mid B[\nu]\}$ through a QUIC edge. However, QUIC graphs as presented here cannot express the reverse inclusions of the form $\{\nu \in \mathrm{F}_1 | B[\nu]\} \subseteq \mathrm{F}_2$. There are

Set Operation		Operation using Union/Intersection
$F_1 \subseteq F_2 \uplus F_3$	$\Leftrightarrow$	$F_1 \subseteq F_2 \cup F_3 \ \wedge \ F_2 \cap F_3 \subseteq \emptyset$
$F_2 \uplus F_3 \subseteq F_1$	$\Leftrightarrow$	$F_2 \subseteq F_1 \ \wedge \ F_3 \subseteq F_1 \ \wedge \ F_2 \cap F_3 \subseteq \emptyset$
$F_1 \subseteq F_2 \setminus F_3$	$\Leftrightarrow$	$F_1 \subseteq F_2 \ \wedge \ F_1 \cap F_3 \subseteq \emptyset$
$F_2 \setminus F_3 \subseteq F_1$	$\Leftrightarrow$	$F_2 \subseteq F_1 \cup F_3$

Figure 7.6. – Encoding set difference and disjoint union in QUIC graphs.

two main reasons for this restriction: (a) Representing reverse inclusions requires a new type of edge relation along with fresh reduction rules for this edge. Additionally, there are many interactions between this new type of relation and existing relations that need to be captured. (b) Reverse inclusions require an abstract domain that implements the underapproximate semantics whereas the inclusions used in QUIC graphs use the standard overapproximate abstract semantics. This ensures that existing abstract domains can be integrated with QUIC graphs without introducing new domain operations. A full theory of QUIC graphs that captures both types of relations will be tackled in the future.

The other expressivity limitation arises from the introduction of union and intersection operations. Note that the relation $F_1 \cup F_2 \subseteq F_3$ can be equivalently expressed simply as $F_1 \subseteq F_3 \ \wedge \ F_2 \subseteq F_3$. Likewise the intersection $F_3 \subseteq F_1 \cap F_2 \ \Leftrightarrow \ F_3 \subseteq F_1 \ \wedge \ F_3 \subseteq F_2$. This motivates the direction of the union and intersection hyperedges in QUIC graphs. Relations between nested unions and intersections are not directly represented unless special existentially quantified variables are permitted in the graph.

Example 7.4. For instance, the relation $(F_1 \cup F_2) \cap F_3 \subseteq F_4$ cannot be expressed unless a special existentially quantified set variable F_5 is introduced with the constraints

$$\dot{\bigcap} F_5 \mathrel{\dot{\subseteq}} \dot{\bigcup} F_1, F_2 \Big|_{\top} \wedge \dot{\bigcap} F_1 \mathrel{\dot{\subseteq}} \dot{\bigcup} F_5 \Big|_{\top} \wedge \dot{\bigcap} F_2 \mathrel{\dot{\subseteq}} \dot{\bigcup} F_5 \Big|_{\top} \wedge \dot{\bigcap} F_5, F_3 \mathrel{\dot{\subseteq}} \dot{\bigcup} F_4 \Big|_{\top}$$

Finally, relations involving disjoint unions and set difference can also be represented directly using QUIC graph as shown in Figure 7.6.

Self Loops: Self-loops on QUIC graphs are quite useful to encode assertions that are true of the contents of F_1 in relation to the scalar program variables $f_1, \ldots, f_n$.

Example 7.5. Let F_1 be a set and f be a program variable. One wishes to express that every element in F_1 is between f and f + 10. It can be done in the QUIC graph domain using the self-loop from F_1 to itself labeled by the assertion $\nu \geq f \ \wedge \ \nu \leq f + 10$. In effect, the loop represents the containment relation written

$$F_1 \subseteq \{\nu \in F_1 \mid \nu \geq f \ \wedge \ \nu \leq f + 10\} \quad \text{or} \quad \forall \nu \in F_1.\, \nu \geq f \ \wedge \ \nu \leq f + 10\,.$$

QUIC graphs naturally represent relationships between set variables, singleton sets and the empty set. However, QUIC graphs do not necessarily represent all possible relationships. The next section shows how to derive other relationships from those already in a QUIC graph.

7.3. Closure

The *closure* of a QUIC graph adds all of the implied logical facts to both the QUIC graph and the base domain. Most of the domain operations of a QUIC graph are defined in terms of the closure by the application of inference rules to add edges to a QUIC graph and strengthen the existing edge labels.

Inference rules are shown in full in Figure 7.7. There are three judgment forms. One states when given a combined domain of a QUIC graph and a base domain, $S = (G, B)$, a particular containment relationship is derivable. If the relationship is derivable, the inference judgment provides a predicate B_e that holds on that relationship. The judgment takes the form

$$(G, B) \vdash \dot{\bigcap} \bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup} \bar{T}^u \Big|_{B_e} ,$$

where $\bar{T}^i$ is the set of intersected vertices and $\bar{T}^u$ is the set of unioned vertices. This judgment relies on an auxiliary judgment $B \vdash \mathrm{f}_1 = \mathrm{f}_2$ where f_1 and f_2 are base domain variables. This judgment states when an equality between variables is derivable from a base domain element (and is supplied by the base domain). There is also a judgment $(G, B) \vdash \mathrm{f}_1 = \mathrm{f}_2$ that states when an equality can be derived from set constraints.

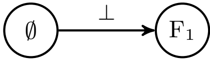
7.3.1. Inference Rules

Following is an explanation of the inference rules for QUIC graphs in detail.

The (Emp) inference rule generates QUIC graph edges from the empty set to any node, labeled with the bottom base domain element $\bot$ (i.e., with the $\emptyset$ concretization or is equivalent to the predicate **false**).

Example 7.6. Consider the QUIC graph G

By applying (Emp), one gets the QUIC graph G':



The (Self-Loop) and (Self-Prop) inference rules generate and strengthen the labels present on self loops in QUIC graphs. The strengthening takes information from an outgoing edge and propagates it back to the self loop.

$$\frac{}{\left(G \ \wedge\ \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_e}, B\right) \vdash \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_e}}\ (\textsc{In-Graph-R}) \qquad \frac{}{(G,B) \vdash \dot{\bigcap}\emptyset \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{\bot}}\ (\textsc{Emp})$$

$$\frac{}{(G,B) \vdash \dot{\bigcap}T \dot{\subseteq} \dot{\bigcup}T\Big|_{\top}}\ (\textsc{Self-Loop}) \qquad \frac{(G,B) \vdash \dot{\bigcap}T \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap}T \dot{\subseteq} \dot{\bigcup}T\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap}T \dot{\subseteq} \dot{\bigcup}T\Big|_{B_a \sqcap B_b}}\ (\textsc{Self-Prop})$$

$$\frac{(G,B) \vdash \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_e}}{(G,B) \vdash \dot{\bigcap}T, \bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_e}}\ (\textsc{Add-Left}) \qquad \frac{(G,B) \vdash \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_e}}{(G,B) \vdash \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}T, \bar{T}^u\Big|_{B_e}}\ (\textsc{Add-Right})$$

$$\frac{(G,B) \vdash \dot{\bigcap}T^i \dot{\subseteq} \dot{\bigcup}T_1^u, \cdots, T_m^u\Big|_{B_0} \qquad (G,B) \vdash \dot{\bigcap}T_j^u \dot{\subseteq} \dot{\bigcup}T_j^u\Big|_{B_j},\ \text{for } j = 1 \cdots m}{(G,B) \vdash \dot{\bigcap}T^i \dot{\subseteq} \dot{\bigcup}T_1^u, \cdots, T_m^u\Big|_{B_0 \sqcap (\bigsqcup_{j=1}^m B_j)}}\ (\textsc{Union-Prop})$$

$$\frac{(G,B) \vdash \dot{\bigcap}T_j^i \dot{\subseteq} \dot{\bigcup}T_j^i\Big|_{B_j},\ \text{for } j = 1 \cdots m \qquad (G,B) \vdash \dot{\bigcap}T_1^i, \cdots, T_n^i \dot{\subseteq} \dot{\bigcup}T_u\Big|_{B_0}}{(G,B) \vdash \dot{\bigcap}T_1^i, \cdots, T_n^i \dot{\subseteq} \dot{\bigcup}T^u\Big|_{B_0 \sqcap (\bigsqcap_{j=1}^m B_j)}}\ (\textsc{Inter-Prop})$$

$$\frac{(G,B) \vdash \dot{\bigcap}\bar{T}_a^i \dot{\subseteq} \dot{\bigcup}T, \bar{T}_a^u\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap}T, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup}\bar{T}_b^u\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap}\bar{T}_a^i, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup}\bar{T}_a^u, \bar{T}_b^u\Big|_{B_a}}\ (\textsc{Union-Trans})$$

$$\frac{(G,B) \vdash \dot{\bigcap}\bar{T}_a^i \dot{\subseteq} \dot{\bigcup}T\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap}T, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap}\bar{T}_a^i, \bar{T}_b^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_a \sqcap B_b}}\ (\textsc{Inter-Trans})$$

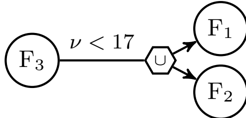
$$\frac{(G,B) \vdash \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_e}}{(G,B) \vdash \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_e \sqcap B}}\ (\textsc{Base-Str}) \qquad \frac{B \vdash \mathrm{f}_1 = \mathrm{f}_2}{(G,B) \vdash \dot{\bigcap}\{\ \mathrm{f}_1\ \} \dot{\subseteq} \dot{\bigcup}\{\ \mathrm{f}_2\ \}\Big|_{\nu = \mathrm{f}_1}}\ (\textsc{Eq-Base})$$

$$\frac{(G,B) \vdash \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_b}}{(G,B) \vdash \dot{\bigcap}\bar{T}^i \dot{\subseteq} \dot{\bigcup}\bar{T}^u\Big|_{B_a \sqcap B_b}}\ (\textsc{Double-Edge})$$

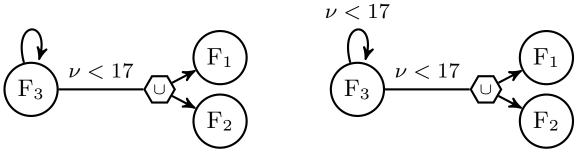
$$\frac{(G,B) \vdash \dot{\bigcap}\{\ \mathrm{f}_1\ \} \dot{\subseteq} \dot{\bigcup}\{\ \mathrm{f}_2\ \}\Big|_{B_a} \qquad (G,B) \vdash \dot{\bigcap}\{\ \mathrm{f}_2\ \} \dot{\subseteq} \dot{\bigcup}\{\ \mathrm{f}_1\ \}\Big|_{B_b}}{(G,B) \vdash \mathrm{f}_1 = \mathrm{f}_2}\ (\textsc{Eq-Set})$$

Figure 7.7. – Inference rules for closure of QUIC graphs. **Notation:** $\bar{T}^i, \bar{T}^u$ are sets of vertices, T are individual vertices of the graph, B, B_a, B_b are base abstract states and x, y are base domain variables.

Example 7.7. Consider the QUIC graph G

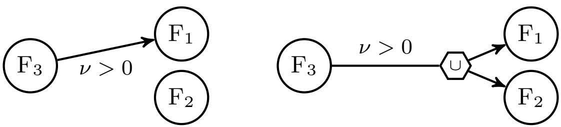


Evaluating the (SELF-LOOP) rule on F_3 gives G' on the left. Evaluating the (SELF-PROP) rule on F_3 and $F_1 \cup F_2$ pushes the predicate $\nu < 17$ onto the self loop at F_3, giving G'' on the right:



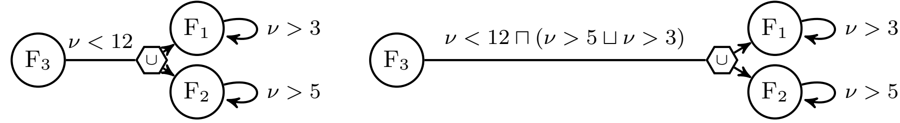
The (ADD-RIGHT) rules allows adding extra elements to the union on the right-hand side of an inclusion. (ADD-LEFT) is the dual rule for intersection.

Example 7.8. Consider the QUIC graph G on the left. Applying (ADD-RIGHT) to F_3 and F_1, adding F_2, gives the QUIC graph G' on the right:



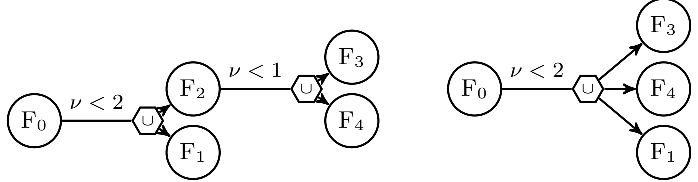
The (UNION-PROP) rule pushes information from self loops backward onto edges. (INTER-PROP) performs the same operation on intersections.

Example 7.9. Consider the QUIC graph G on the left. Applying (UNION-PROP) to F_3, F_1 and F_2 yields the graph shown to the right.



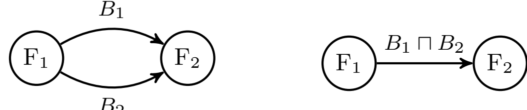
The (UNION-TRANS) rule combines two union edges to produce a single union edge. This rule loses information from one of the edges, but that information can be regained through the application of (UNION-PROP). The symbols $\sqcap$ and $\sqcup$ are used for the meet and join operator in the base domain, respectively. (INTER-TRANS) does the same for intersection without losing information.

Example 7.10. Consider the QUIC graph G on the left. The two union edges are combined to produce the union edge on the right. Even though $\nu < 1$ is a stronger constraint than $\nu < 2$, the resulting constraint is the weaker $\nu < 2$.



The (Double-Edge) rule merges two edges between the same vertices into a single edge. QUIC graphs do not track multiple edges between the same two vertices, so a duplicate edge must immediately be converted to a single edge with this rule.

Example 7.11. Consider the two edges on the left. Since QUIC graphs cannot represent these, they are combined into the single edge on the right.



The rule (Base-Str) strengthens any edge in the graph with the current facts from the base domain B. The rule (Eq-Base) strengthens relationships in the set domain by adding a constraint on the bound variable. The latter also uses equality in the base domain to infer equalities in the set domain. Oppositely, (Eq-Set) uses equalities in the set domain to infer base domain equalities.

Definition 8 (Closure). *Let (G, B) be a QUIC graph and a base domain predicate. The closure (G^*, B^*) is the conjunction of all*

$$\dot{\bigcap}\bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup}\bar{T}^u\Big|_{B_e} \quad \textit{such that } (G, B) \vdash \dot{\bigcap}\bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup}\bar{T}^u\Big|_{B_e}$$

and the constraining of B with all equalities $\mathrm{f}_1 = \mathrm{f}_2$ given by the judgment $(G, B) \vdash \mathrm{f}_1 = \mathrm{f}_2$.

7.3.2. Soundness

First, soundness for systems of inference rules must be defined. For a QUIC graph analysis to be sound, the underlying system of inference rules must be sound.

Definition 9 (Inference Soundness). *An inference is sound if the following two conditions hold:*

1. *if $(G, B) \vdash \dot{\bigcap}\bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup}\bar{T}^u\Big|_{B_e}$, then $\gamma((G, B)) \subseteq \gamma\left(\dot{\bigcap}\bar{T}^i \mathrel{\dot{\subseteq}} \dot{\bigcup}\bar{T}^u\Big|_{B_e}\right)$.*
2. *if $(G, B) \vdash \mathrm{f}_1 = \mathrm{f}_2$, then for all $(\eta, \eta_B) \in \gamma((G, B))$, it is that $\eta_B(\mathrm{f}_1) = \eta_B(\mathrm{f}_2)$.*

Let us assume that $\widetilde{B}$, the base domain, is a sound abstract domain [CC77].

Theorem 6 (Inference Soundness). *The inference rules in Figure 7.7 are sound according to Definition 9.*

7.3.3. Complexity

Closure of a QUIC graph is potentially expensive since the number of edges in the closure can be exponential in the worst case.

Theorem 7 (Inference Complexity). *There are $O(2^n)$ possible hyperedges in a QUIC graph with n vertices.*

Without "tactics" to apply the rules cleverly in an implementation, the inference over QUIC graphs is intractable.

7.3.4. Lazy Inference Implementation

Now is discussed how the inference operation is implemented with the QUIC graph approach. The goal of the implementation is to avoid a blowup in the number of graph edges and running time each time a closure is to be computed. Lazy inference is a tactic that computes an effective closure on demand. It is composed of many strategies. Here, the most important concepts used in our implementation are described.

1. *Simplification:* many simplification passes keep the QUIC graph in a canonical form. This automatically takes into account many of the inference rules from Fig. 7.7.
2. *Lazy inference:* Instead of computing the closure eagerly and adding a set of extra edges to the graph, QUIC graphs do so lazily whenever edge membership queries are issued by the abstract domain.
3. *Partial closure:* Note that many of the edges that are generated by a closure are not necessarily useful as invariants for proving properties. Therefore, QUIC graphs uses heuristics that choose edges to query. This process is called *candidate generation* since it affects which invariant candidates are considered by our analyzer at each step.

Simplification: Simplification consists of many different parts. The first simplification deals with edges from the empty set $\emptyset$. As such, they do not contribute to the inference. It is assumed that these edges implicitly exist but do not represent them.

Next, simplication considers *equivalence classes* of set variables. Two sets $\mathrm{F}_1, \mathrm{F}_2$ are equivalent if $\mathrm{F}_1 \subseteq \mathrm{F}_2 \ \wedge \ \mathrm{F}_2 \subseteq \mathrm{F}_1$. Equivalence classes are identified using a maximal strongly connected component algorithm on the QUIC graph. Equivalence classes of sets can be compacted and one representative is chosen using a pre-defined variable ordering. All membership queries involving members of equivalence classes are first rewritten in terms of the representative members of the classes.

The (Double-Edge) rule is implicitly implemented by the data structure whenever there is an attempt to add two edges between the same set of nodes. Finally, the (Self-Loop), (Self-Prop), (Union-Prop) and (Inter-Prop) rules are used to propagate labels and add new edges between representatives of equivalence classes. These rules also strengthen the labels on edges.

Lazy Inference: Next, inference on demand is implemented by applying the inference rules to decide if a queried edge is present in the graph. This is performed by iterating the (Union-Trans) and (Inter-Trans) rules to compute transitive closures.

Candidate Generation: The computation of a lazy inference is driven by the choice of candidate query edges that might be added to the graph. To this end, a candidate generation heuristic is used in the implementation to choose candidate invariant facts. There are many possible heuristics for generating candidate query edges. The implementation uses set expressions that appear in the program including properties to be proved as a source of edges to keep in the partial closure. Another choice includes edges that are generated through transfer functions such as assignments. Once generated, an edge is kept as a candidate edge for future inference computations.

7.4. Domain Operations

This section discusses the abstract domain operations over the reduced product domain of QUIC graphs and the base domain $\widetilde{B}$ for base domain variables.

Notation: Let G be a QUIC graph. The notation $G[\mathrm{F}_1 \leftarrow \mathrm{F}_0]$ denotes the graph obtained by changing the label of vertex F_1 to F_0. The notation is extended to set expressions so that $T[\mathrm{F}_1 \leftarrow \mathrm{F}_0]$ denotes the substitution of F_1 by F_0 for each occurrence in the expression T.

Abstract domain transition functions are defined using semantic functions:

$$[\![\mathrm{stmt}]\!]^{\mathrm{S}} : \widetilde{S} \to \widetilde{S}$$

These functions are parameterized by stmt, which is a command in the language of sets and base domain operations. It takes an abstract state $S = (G, B) \in \widetilde{S}$ composed of a graph G and a base domain element B and returns an abstract state S' that represents the state after having executed command stmt on S.

Simple Transfer Functions: The transfer functions for some basic assignment states are represented below. In each case, the result may not be closed. Therefore, inference may be applied on the result, if necessary.

$$[\![\mathbf{havoc}\ \mathrm{F}_1]\!]^{\mathrm{S}}(G,B) \stackrel{\text{def}}{=} (G[\mathrm{F}_1 \leftarrow \mathrm{F}_0], B) \qquad \mathrm{F}_0 \text{ is fresh}$$

$$[\![\mathrm{F}_1{:=}\emptyset]\!]^{\mathrm{S}}(G,B) \stackrel{\text{def}}{=} \left(G[\mathrm{F}_1 \leftarrow \mathrm{F}_0] \ \wedge\ \mathrm{F}_1 \xrightarrow{\ \bot\ } \emptyset\ , B\right) \qquad \mathrm{F}_0 \text{ is fresh}$$

$$[\![\mathrm{F}_1{:=}T]\!]^{\mathrm{S}}(G,B) \stackrel{\text{def}}{=} \left(G[\mathrm{F}_1 \leftarrow \mathrm{F}_0] \ \wedge\ \mathrm{F}_1 \rightleftarrows T\ , B\right) \qquad \mathrm{F}_0 \text{ is fresh}$$

The command **havoc** F_1 assigns F_1 to a non-deterministic value. Rather than projecting the vertex F_1 from the graph, it renames the existing vertex to a fresh variable F_0. The

vertex F_0 remains in the graph as a *history variable*. Operations such as join and widening will eliminate the necessary history variables, ensuring that they do not propagate out of scope. However history variables will exist for as long as possible as this may allow additional relationships to be inferred.

The command $F_1{:=}\emptyset$ assigns F_1 to the empty set. Because it is performing a destructive update to F_1, F_1 is renamed to a history variable F_0 as is standard when performing a destructive update. This leaves the symbol F_1 completely unconstrained so that when constraints are added to F_1, those are the only constraints on F_1. The added constraint here is $F_1 \subseteq \emptyset$, labeling it with the strongest possible predicate $\bot$.

The command $F_1{:=}T$ assigns a set element T to F_1. This creates the two edges representing both $F_1 \subseteq T$ and $T \subseteq F_1$. The edge labels are set to $\top$ and thus not shown as all the information from B can be added to these edges through the inference procedure.

Meet (Intersection): The meet of two abstract states $(G_1, B_1) \sqcap (G_2, B_2)$ is the conjunction of the set constraints and meet in the base domain (i.e., $(G_1 \wedge G_2, B_1 \sqcap B_2)$ where the $\sqcap$ notation is overloaded for both the QUIC graph and the base domain). Note that viewing the set constraints as graphs, meet is the union of two graphs.

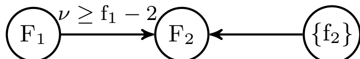
Set Assignment Rule: The domain operation for assignments of the form $F_1{:=}T_1 \cup T_2 \cdots \cup T_n$ (similarly for $F_1{:=}T_1 \cap \cdots \cap T_n$) builds upon the already defined operations. The basic idea is to replace F_1 by a history variable F_0 and introduce hyper-edges to capture the new relations formed. For simplicity, consider the case $n = 2$

$$[\![\mathrm{F}_1{:=}\mathtt{T}_1 \cup \mathtt{T}_2]\!]^{\mathrm{S}}(G, B) \stackrel{\text{def}}{=} \left(G[\mathrm{F}_1 \leftarrow \mathrm{F}_0] \;\wedge\; \text{[graph: } \mathrm{F}_1 \text{ — } \cup \text{ — } T_1[\mathrm{F}_1 \leftarrow \mathrm{F}_0],\ T_2[\mathrm{F}_1 \leftarrow \mathrm{F}_0]\text{]}\ ,\ B \right)_{\mathrm{F}_0 \text{ is fresh}}$$

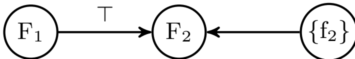
Intersection, disjoint union and set difference operations are similar. They rename F_1 to a fresh variable F_0 and rename T_1 and T_2 similarly, if appropriate. Then a constraint that represents the appropriate equality is added to the QUIC graph.

Base Domain Assignment: An assignment to the base domain variables $f_1 := e$ will result in three changes: (a) applying the assignment to the base domain element B, (b) applying the assignment to each edge label in the QUIC graph G and (c) any singleton node in the graph that involves f_1 needs to be updated either by computing its post-condition w.r.t to the assignment, if invertible or renamed to a fresh set variable F_0 for a destructive assignment. All applications invoke the base domain transfer function and thus rely on the base domain for introduction (or not) of history variables. This is illustrated through a simple example.

Example 7.12. Consider the QUIC graph G



and let $B : f_2 \geq f_1$. Consider the destructive assignment $f_1 := f_2 + 1$. The transfer function yields the QUIC graph G':



with the assertion $B' :\ f_1 = f_2 + 1$. A partial closure on the result is computed, which effectively pushes the constraint $f_1 = f_2 + 1$ on the edges of the graph G'.

Choose: The **`choose`** command selects an element from a set and assigns it to a base domain variable. It takes quantified information from the set domain and applies it to the resulting base domain variable. The strategy to handle $f_1 :=$ `choose`(T) for an abstract state (G, B) is the following:

1. Perform an inference operation on (G, B) giving (G^*, B^*).
2. Extract the base domain constraint B_e from a self-loop on T:

$$G^* = G' \ \wedge \ \dot{\bigcap} T \dot{\subseteq} \dot{\bigcup} T \Big|_{B_e} .$$

3. Replace the bound variable ν in B_e with a fresh base domain variable f_2 giving B_2. This process transfers all the facts that apply to elements in set T and applies them to the variable f_2.
4. Compute the meet $B' = B^* \sqcap B_2$. This transfers those facts about f_2 to the base domain.
5. Perform the destructive update $f_1 := f_2$ on (G^*, B') to get the result of choose.

Projection: The projection of a base-domain variable f_1 from (G, B) is performed by (a) projecting f_1 from B and (b) projecting f_1 from each label in G. These are performed by calling the projection defined in the base domain $\widetilde{B}$.

The projection of a vertex T from the QUIC graph G first computes its partial closure (G^*, B^*). Next, all conjuncts involving the vertex T are removed from G^* to obtain the projection.

Join: Let (G_1, B_1) and (G_2, B_2) be the arguments for the join operation. First, the partial closure of (G_1^*, B_1^*) of G_1 and likewise the partial closure (G_2^*, B_2^*) of (G_2, B) are computed. The join (G, B) is then defined where $B = B_1^* \sqcup B_2^*$ and G is all conjuncts

$$\dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_{B_1 \sqcup B_2}$$

where there exists some G'_1 and G'_2 such that

$$G_1^* = G'_1 \ \wedge \ \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_{B_1} \quad \text{and} \quad G_2^* = G'_2 \ \wedge \ \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_{B_2} .$$

Widening: As such the QUIC graph domain is a product of a finite graph domain and an abstract base domain. Widening is required iff the base domain does not satisfy the ascending chain condition. The basic widening algorithm is precisely the same as the join operation with the modification that the base domain widening operation is applied for each QUIC edge instead of widening.

7.5. Evaluation

The section presents a preliminary evaluation of the prototype analyzer. The QUIC graphs domain introduced in this paper has two main aspects: (a) it enables relational reasoning between sets to prove that one set (expression) is contained in another; and (b) it allows us to qualify relations between sets using base domain predicates, in effect allowing us to reason with *set comprehension*. The evaluation in this section is intended to answer the following questions:

1. How much does each of the two ingredients (relations between sets + set comprehensions) add to the ability of the analysis to prove properties of commonly encountered use cases?

2. What is the added cost due to each of the two ingredients to the overall domain?

For comparison in the evaluation, there are two simplified versions of the QUIC graphs domains: the 'set' and 'elem' domains. (A) The 'set' domain allows relations between sets but no comprehensions. This is a realization of a container-as-a-whole approach. This domain is created using the trivial two element $(\bot, \top)$ base domain. (B) The 'elem' domain disallows relations between sets but allows us to reason about the contents of the set using a *summary variable*. This is a realization of a content-centric domain. To simulate this domain, the original QUIC graphs domain is modified to just allow self loops on nodes as the only possible edge. In effect, the predicate on such an edge must be true of every element in the set. Furthermore, the process is exactly equivalent to introducing a *summary variable* for each set variable and performing a base-domain analysis using this summary variable.

Benchmarks: The next step is to choose a series of benchmarks that represent common motifs for set (container) usage in dynamic languages. To evaluate the approach two sets of benchmarks were used. The analysis was designed using the first set of benchmarks, which exercise four commonly occurring operations on containers 'copy', 'filter', 'partition' and 'merge'. Next, the analysis was run, *unmodified*, on translated versions of all of the programs from the Python test suite [Pyt12] for dictionaries and sets. Extraneous parts of these tests were removed and simply the core part of each program was translated to

an equivalent program in the input language. Each test has a set of pre-defined assertions to be established by the analyzer.

Table 7.1. – Results on a set of small benchmarks. **Base Vars**: # of base domain (numerical) variables, **Set Vars**: # of set variables, **Num Prp**: # of assertions to be proved, **T**: Time taken (seconds), **#I**: number of iterations of abstract interpreter before convergence. – represents a time out (600 seconds)

	Base	Set	Num	# Proved			Time Taken (Iterations)					
ID	Vars	Vars	Prp	QG	set	elem	QG_T	#I	set_T	#I	$elem_T$	#I
copy	1	6	2	2	2	0	0.2	2	0.2	2	0	2
filter	4	6	2	2	1	0	0.6	3	0.5	3	0.1	2
generic_max	3	8	6	3	0	0	0.9	6	0.6	6	0.2	4
merge	2	14	2	1	1	0	0.6	4	0.6	4	0.1	4
partition	4	8	4	4	2	0	1.1	3	0.9	3	0.2	2
b_filter	6	6	2	2	0	0	0.7	3	0.6	3	0.1	2
b_map	9	7	2	2	2	2	0.2	5	0.3	5	0.1	4
b_max_min	3	4	1	1	1	1	0.4	3	0.3	3	0.1	2
b_reduce	7	4	1	0	0	0	0.4	3	0.3	3	0.1	2
iter_ind	20	12	1	1	0	0	84.4	39	67.9	39	6.8	14
mul_ret	9	2	2	2	0	0	0.2	6	0.1	6	0.1	6
nest_dep	5	7	1	0	0	0	2.2	12	2.2	12	0.4	6
resize1	15	5	5	4	0	0	1.7	18	1.1	18	1	18
simp_cond	11	5	4	3	0	0	4.6	12	1.6	12	1.3	12
simp_nest	9	10	2	0	0	0	–	1399	–	1612	0.7	6
srange	6	2	2	2	0	0	0.1	6	0.1	6	0.1	6
Total			37	29	9	3	98.3	125	77.3	125	11.4	92

Results: Figure 7.1 summarizes the results of the analysis run on these benchmarks on an Apple MacBook Pro, on a 2.2GHz Intel Core i7 with 8GB RAM running Mac OS X 10.8.2. Now, the precision and running time are compared. The memory required by most analysis runs was under 150 MB. It is quite clear from the results table that the combination of relational reasoning and comprehension using base domain predicates is quite powerful. Whereas the QUIC graphs domain can prove a majority of the properties, restricting it either by removing the comprehensions (set) or removing the relations between sets (elem) are both able to prove much fewer properties. Furthermore, every property proved by these domains is also proved by the QUIC graphs domain.

The comparison of costs indicates that the QUIC graphs domain is 1.2× slower than the set domain. However, it is 9× slower than the elem domain. The difference in performance is entirely expected since the QUIC graphs domain has to perform a lot more reasoning steps. Additionally, one example times out (after 600 seconds).

7.6. Limitations.

Precision While QUIC graphs are an effective abstract domain, but some properties were not proven due to imprecision in the analysis. There are four sources of this imprecision: (1) incomplete candidate generation, (2) imprecise base domain, (3) no cardinality reasoning, and (4) syntactic restrictions within QUIC graphs.

To reduce needless inference in many examples, QUIC graphs uses candidate generation (Section 7.3) to reduce the number of rule applications. Because candidate generation reduces the potential edges that can result from a join, it can cause the join to lose more information than is strictly necessary. This is this cause for many of the failures in Table 6.2, including the failure to prove one of the properties in 'merge'. Further work on candidate generation is quite important.

Because the base domain is also an abstract domain, it is imprecise and may not be able to represent some necessary relationship. This is especially the case when there is a transformation applied to all elements of a set. The base domain must be able to represent that transformation that occurs to each element as a relation. In this test suite there is only one test that exercises this ability and the relations are all representable as linear relationships, so this imprecision does not affect the results. However, if this were a problem, a new base domain could be selected because QUIC graphs are agnostic to the base domain.

The QUIC graphs domain does not track the cardinality of sets beyond empty, singleton, and unknown. As has been previously shown [Kun07], cardinality can strengthen relationships, and therefore in QUIC graphs, cardinality constraints would create additional closure rules. For example, if for some set F there is the constraint that $\{1,3,7\} \subseteq \mathrm{F}$ and that $|\mathrm{F}| = 3$, then $\mathrm{F} \subseteq \{1,3,7\}$ can be inferred. It is possible that cardinality information could provide sufficient information to prove properties that failed in this test suite, but this information could likely be inferred in another way (such as better candidate generation) because most sets in the test suite have unknown cardinality.

QUIC graphs are syntactically restricted to allow comprehensions only on one side of a subset relationship. Reverse inclusions (Section 7.2) are not supported. It is expected that the ability to know that an element exists in a set will be beneficial when abstracting other containers using sets.

Scalability Despite all of the techniques here, the theoretical complexity of computing a join or inclusion check in QUIC graphs has not improved beyond exponential. In practice, this may not be a problem with appropriate use of additional lazy techniques, as used in satisfiability solvers. However, with current implementations, the sometimes exponential cost can be a detriment to an analysis.

As a result of the complexity of the join, even with the candidate generation techniques, there are scalability problems when the number of symbols increases. This was seen in the application of QUIC graphs to the HOO abstract domain for objects. When the number of variables increased, the performance worsened significantly. However, there are approaches that may improve scalability beyond what QUIC graphs has today.

First, scalability could be improved by tracking equalities separately from the set

domain. In practice, there are many constraints of the form $F_1 = F_2$ and these simply make all computations more difficult. They introduce many more ways inference can happen and they introduce many more join possibilities. Alternatively, these could be tracked with a disjoint union data structure that maintains equivalence classes. Only the representative from each equivalence class would be present in the QUIC graph. For situations when many equalities are used, this could eliminate many redundant cases that could cause a divergence in the analysis.

Second, scalability could be improved by only keeping track of relations between symbols that are explicitly formed in the program. To do this, multiple domain instances would be used for each abstract state. Each domain instance would keep track of a subset of all of the set symbols. By doing this, the overall number of symbols per domain goes down. Since the exponential behavior is of the form 2^n, dividing that into m different parts, produces a complexity of $m \cdot 2^{\frac{n}{m}}$, which is significantly better because the exponential is reduced in exchange for a simple multiplication.

7.7. Related Work

There exists a large number of container analyses, mostly focused on arrays. Although there are many different approaches, the problem is fundamentally the same: partitioning an array in order to summarize different segments. Gopan et al. [GRS05], Halbwachs et al. [HP08] and Cousot et al. [CCL11] use an abstract interpretation framework with materialization and summarization. Therein, the partitions are inferred from the structure of the program. Seghir et al. [SPW09] perform this in the context of predicate abstraction, similarly to abstract interpretation. Jhala et al [JM07], McMillan [McM08], Kovacs et al. [KV09], and Dillig et al. [DDA10, DDA11] use theorem provers to perform this partitioning. Our approach does not use a partitioning scheme except for the special case of loops that iterate over sets. Furthermore, these approaches do not, in general, reason about comprehensions or relate the contents of different arrays.

There are several alternative approaches to reasoning about container manipulations. Marron et al. [MSHK07, MMLH+08] used a shape analysis to emulate data storage of containers. They used appropriate inductive predicates with carefully tuned, simplified implementations of the containers to get an automatic analysis. Dillig et al. [DDA11] extended their previous work on arrays to more generic containers. Their approach uses base domain predicates as constraints on the sets of keys for maps. This is a highly tuned example of what a content-centric domain. Their approach does not directly infer relationships between containers. However, they can indirectly infer relations through data invariants that relate their contents. Finally, Pham et al. [PTTC11] introduced a relational domain for sets. Their domain is similar to QUIC graphs in that it is designed to directly represent relations between sets. Their approach represents the as-a-whole approach for the most part. It does support a base domain of uninterpreted functions and can be precise for a restricted class of programs. Because they support only uninterpreted functions for the base domain, they have been able to implement some under-approximations required to infer equalities with predicate comprehensions, but this

base domain does not support any manipulation or reductions and thus is weaker than domains that QUIC graphs supports.

The invariant generation procedure of [GMT08] could infer many of the invariants that QUIC graphs infers given a sufficiently expressive list of predicate templates. They select from templates to use for quantified facts. As a result, their analysis requires user input and guidance for success, but the approach does offer some additional generality. Bouajjani et al. [BDES12] present a similar, more automatic approach to dealing with quantified invariants, by pre-selecting appropriate templates for many applications. They apply their work to linked list structures and support multiple bound variables to be able to maintain sortedness properties. Like the work of [MSHK07], they use a shape analysis framework to approximate the shape and data of lists, while maintaining quantified side conditions on an integer base domain.

The QUIC graph data structure is similar to a formalization of constraint graphs [AFFS98, Fla97] use to prove complexity of satisfaction of constraints [AKVW93]. While the encoding is similar, there is no need for base domain labels since constraint graphs are unable to place quantified restrictions on the contents of the sets they constrain. In general, constraint graphs do represent sets, but they are intended to use sets to analyze programs rather than analyzing set-manipulating programs.

The decision procedures community has largely solved this problem of relational containers, but only for the problem of entailment checking. Decision procedures do not perform invariant generation. Bradley et al. [BMS06] demonstrated decision procedures for arrays and other containers. The Z3 SMT solver implements an optimized version [dMB09] of these decision procedures to speed up these problems. Also, Lam et al. [LKR05] and Kuncak [Kun07] developed a system that simultaneously reasons about sets and their cardinalities relationally. Since these tools solve the decision problem rather than the inference problem, they are incomparable, however the optimizations used in [dMB09] are similar to operations that are defined in QUIC graphs closure because they are Boolean algebra-like operations.

7.8. Summary of QUIC Graphs

This chapter has demonstrated a relational abstract domain for sets that combines a content-centric analysis with a container-as-a-whole approach. This is achieved through a new representation for set constraints called QUIC graphs that simplifies the representation of set expressions and inclusion relations that use comprehensions. Our evaluation of this domain shows that a combined approach using QUIC graphs is quite effective in practice. It outperforms weaker alternatives such as a content-centric approach and a container-as-a-whole approach.

By integrating the QUIC graphs domain with a heap domain such as HOO, set-based abstractions have opened up a wide range of new abstractions for heaps and containers. By mixing content-centric and as-a-whole reasoning, dependent domains such as HOO are able to reason about both individual values as well as completely unknown sets of values, making the resulting analysis both more efficient and more flexible.

8. Conclusions and Future Work

In this dissertation, I have presented a different way of tackling the dynamic language verification problem. Rather than approaching whole programs once they have been completed, the approaches presented here are targeted at libraries that are under development and have not yet been deployed. These techniques can allow library developers to understand and debug their code before it is deployed, avoiding the problems of late bugs that are found after deployment and thus saving money, reducing risk, and creating better relationships with clients who use their libraries.

To be able to analyze library code and other partial programs, this dissertation presented the following thesis statement:

The combination of local heap reasoning with sets provides a means to construct direct, parametric abstractions suitable for automatically analyzing dynamic language libraries.

This dissertation developed a heap abstraction (Chapter 4) that supported local reasoning with separation logic. This abstraction is based on an abstraction for sets (Chapter 7) that provides a way represent relationships that form between the unknown objects (Chapter 4) and calls of unknown functions (Chapter 5) that arise when analyzing libraries. The four main contributions that support this these are:

1. ***Heap with Open Objects*** – The HOO abstraction embodies the core of the thesis statement. It is a separation-logic-based abstraction and therefore it supports local heap reasoning via the standard separation logic frame rule. Its use of sets to represent abstract addresses, abstract values, and most importantly abstract attribute partitions gives it the expressiveness to represent the incremental changes to objects that occur when adding, removing, and iterating over open objects. Because it is parameterized by an abstraction for sets, it is a tunable abstraction that can be tailored to suit a variety of different applications.

   By itself, HOO supports the thesis statement. If the purpose of the analysis is to find assertion failures in dynamic language libraries that do not have clients, even when those libraries manipulate objects that were unknown inputs, HOO is well suited to the task.

2. ***Desynchronized Separation*** – If the library code also includes functions as input or in the starting heap, HOO is insufficient. As many dynamic language libraries

do this, it is often necessary to use desynchronized separation. Desynchronized separation builds upon a separation-logic-based analysis and thus is also a local heap abstraction. It partitions the heap into the part that can be modified or accessed by the unknown function and parts that cannot be modified or accessed by the callback function.

3. ***Attribute/Value Trackers*** – Attribute/value trackers extend an abstraction for open objects or containers to improve precision when attributes/values are copied from one object to another. This relies upon the partitioning of attributes that is provided by the underlying abstraction. For HOO this partitioning is provided by the set abstraction. Consequently, attribute/value trackers depend on the set abstraction.

4. ***QUIC graphs*** – QUIC graphs provide a flexible set abstraction. Prior to QUIC graphs very little work had been done on relational abstraction for sets. No work had been done on abstractions that simultaneously considered the contents of sets along with relations between sets. QUIC graphs utilizes a domain for values to provide abstractions of set contents (content-centric reasoning) combined with a relational abstraction for sets (as-a-whole reasoning) to get a result that is better than each of two parts alone. Since objects in dynamic languages have attributes that are constrained both by relationships with other objects and in their contents, QUIC graphs is ideally suited for use with HOO and the analysis of dynamic languages.

By combining the above four abstractions, the analysis provides a number of significant benefits including (1) native relational open object support, (2) local heap abstraction, (3) strong updates, (4) unknown function abstraction, and (5) flexibility. As a result, it provides a general framework for abstractions that can be used for partial dynamic language programs.

The first significant benefit is native open object support. The HOO abstraction does not build upon existing abstractions of objects in non-dynamic languages. Instead, it directly handles the issue of dynamically adding, removing, and iterating over attributes of objects by representing them in a summarized form using sets.

Sets are the central element of native open object support in HOO. Unknown attributes of an open object are represented by partitioning the potentially unbounded number of attributes into a finite number of sets of attributes. These sets of attributes can then be related to other sets of attributes in other objects using an abstraction for sets. This abstraction for sets is a parameter for constructing HOO, so it can be tuned.

The second benefit is local heap abstraction. Local heap abstraction is critical for analyzing libraries. Libraries will not affect the entire heap because libraries are always run with a client that also allocates memory. In the absence of that client, it is critical to bound the behavior of the library to the portion of the heap that it can affect. Because HOO is based on separation logic, it naturally provides this local heap abstraction and thus an analysis performed with HOO can be composed with an analysis of code that affects a different part of the heap transparently.

The third benefit is strong updates. HOO provides strong updates in two forms: at the object level and at the attribute level. At the object level, before an object is read or updated, if the object being referenced is a summary, a single object and a new summary object are split out of that object. At the attribute level, before an attribute is read or written, if the attribute being referenced is a summary, a single attribute and a new summary attribute are split out of the old summary. Furthermore, using attribute/value trackers, if materialized attributes are solely copied from object to object, the analysis can be completely precise. Using the power of the set abstraction, when HOO materializes a single object or attribute from a summary object or attribute, that split that occurs is encoded into the set abstraction, giving the potential to not only undo the operation, but to know exactly how one set was derived from another. This is useful for all varieties of precise reasoning about objects and the heap.

The fourth benefit is unknown function abstraction. Building upon local heap abstraction, desynchronized separation provides a means for reasoning about the behavior of code that calls a function, even if the function that is called is unknown. It splits the heap and records an optimistic view of the portion of the heap that may be modified by the function along with the function that modifies it. This allows desynchronized separation to represent the heap in an intuitive way where the portion of the heap that may have been affected by the function call is represented as what it is: the portion of the heap that may have been affected by the function call.

The final benefit is flexibility. While flexibility does not play a direct roll in analyzing any given JavaScript or other dynamic language program, it is critical for making general frameworks that are applicable across a wide range of programs and a wide range of languages. The HOO abstraction is a mechanism for reasoning about dynamic language programs. It is parameterized by a set abstraction, which means that all of the relational reasoning that HOO provides can be tuned for both performance and precision by selecting an appropriate set abstraction. Additionally, extra values can be trivially added to the abstraction by exchanging the underlying value abstraction without changing anything else.

Further, desynchronized separation is flexible because it is parametric with respect to a separation-logic-based abstraction. Existing abstractions can be extended with support for callback functions by extending them with desynchronization. Similarly, attribute/value trackers are flexible because they can be added to any partition-based abstraction for containers or open objects. These extensions provide flexibility by extending existing abstractions with support for new operations and improved precision.

A final and closely-related contribution of this dissertation is the creation of a flexible abstraction for sets. Because the abstractions for objects and the heap depend so heavily on an abstraction for sets, it is important to have a general, flexible abstraction for sets that combines the ability to reason about contents of sets simultaneously with the ability to reason about relationships between sets as-a-whole. The QUIC graphs abstraction provides exactly this. It provides the rules for combining content-centric and as-a-whole reasoning in a way that allows automatically verifying programs that manipulate sets and contents. It is parameterized by an abstraction for values that is then transformed into an abstraction for sets of those values.

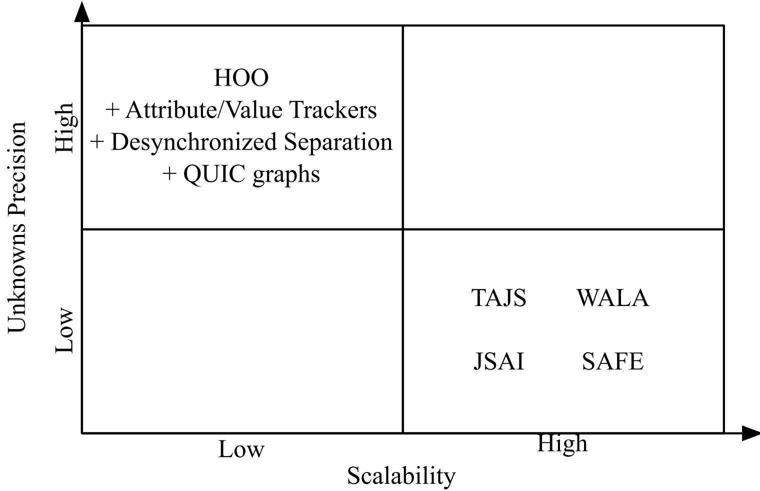


Figure 8.1. – Analysis landscape comparing scalability and unknown attribute name and function precision. The abstractions presented in the dissertation are in a completely different precision/scalability class than existing analyses.

There are actually two classes of static analyzers for dynamic languages as Figure 8.1 shows. Whole-program analyzers such as TAJS, WALA, JSAI, and SAFE focus on scalability and attempt to add enough precision to prove properties about programs when the whole program is there. These analyzers do not require high precision for unknowns much of the time because, with the full context provided by the client code, sufficient precision is possible with simpler non-relational abstractions for unknowns. The abstractions presented in this dissertation occupy a completely different position in the space of analyzers. They are precise first. They are focused on the challenging problems of unknown open objects and unknown function calls that arise in the verification of libraries in the absence of clients. As a result, for now, they do not scale well. Despite this, they provide tools to verify small but complex libraries written in dynamic languages. Furthermore, there are many possible methods to explore to move towards high precision for unknowns and increased scalability.

Limitations and Possible Future Directions As Figure 8.1 shows, there is an obvious possible improvement to the work presented here. The upper-left corner of analyses that are scalable, yet precise for unknowns is conspicuously empty. The reason for this is that it is not yet clear how to make this analysis scalable. However, there are a number of ways in which the analysis could be made faster.

It is possible to take more advantage of the significant number of string constants that are commonly available in JavaScript programs. While input objects may be fully unknown and require the use of HOO to precisely analyze, internal objects that are constructed as part of object-oriented programming may be largely constant and thus not require the full generality of HOO. Handling such objects with a specialized abstraction

could improve performance without sacrificing precision. The trick to accomplishing this would be creating a mechanism for promoting an object that uses the specialized abstraction to the more general HOO abstraction if unknown attributes start being used with the object.

Furthermore, to improve the general HOO, there are innumerable different set abstractions that could be used. I presented QUIC graphs, which is a general framework for designing set abstractions that simultaneously reason about set contents and set relationships. This approach achieves a reasonable level of efficiency by lazily inferring derived relationships between set symbols and values. However, lazy inference does not solve the problem of computing join and widening. The techniques presented here use heuristics for candidate generation where each candidate can be then checked using lazy inference. This approach is neither as precise nor as scalable as possible for many applications.

Regardless of the optimality of QUIC graphs as implemented, it provides an effective basis for sharing information between an abstraction for values and an abstraction for sets. By removing features from QUIC graphs and then specially optimizing it for particular applications, it may be possible to get it to significantly better performance and precision. An example of this would be disabling much of the content-centric reasoning, leaving mostly as-a-whole reasoning, which can be performed for efficiently using constraint solvers. Only when significant value reasoning is required is the full power of QUIC graphs needed.

Additionally, there are many ways in which the analyzer itself could be improved. Currently support for JavaScript is limited. Many basic language features such as arrays and numbers are left unimplemented as they were not needed to demonstrate the abstractions. Practically, these things must be supported. In addition, many control structures such as exceptions and different varieties of loops are not supported. To widely support most real JavaScript, support for these structures is necessary.

Furthermore, there are ways in which both precision and performance may be able to be increased. The HOO abstraction is incredibly flexible. This comes with some good parts and bad. By exchanging the underlying set domain and the join and widening heuristics, it is possible to completely change the behavior and expressivity of the abstraction. The join and widening heuristics determine which objects get merged into summaries and which partitions of objects' attributes get merged into summaries of partitions. Throughout this work, we have utilized allocation site information to guide this summarization. However, it is likely that a variety of different useful heuristics exist.

Finally, with appropriately tuned abstractions, and possibly through the use of techniques such as bi-abduction [CDOY11], it is possible that fully modular analysis for dynamic languages could be realized. With such analyses it may be possible to scale beyond the 1000 to 10000 line programs that are currently possible with whole program analyses, all while maintaining better precision. Of course realizing such goals will require significant engineering and additional research. Regardless, the HOO abstraction, desynchronized separation, attribute/value trackers, and QUIC graphs pave the way for such advances.

Bibliography

[AFFS98] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. A toolkit for constructing type- and constraint-based program analyses. In *Types in Compilation*, pages 78–96, 1998.

[AKVW93] Alexander Aiken, Dexter Kozen, Moshe Y. Vardi, and Edward L. Wimmers. The complexity of set constraints. In *CSL*, pages 1–17, 1993.

[BAT14] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *ECOOP*, pages 257–281, 2014.

[BB01] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1), January 2001.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207, 2003.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

[BCF+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. In *POPL*, pages 87–100, 2014.

[BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[BCLR14] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. $\mathsf{SAFE}_{\mathsf{WAPI}}$: Web api misuse detector for web applications. In *FSE*, 2014.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.

[BDES12] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, pages 1–22, 2012.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.

[BP88] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Trans. Softw. Eng.*, 14(10), October 1988.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[BR06] Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, pages 221–239, 2006.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[Car04] Luca Cardelli. Type systems. In *CRC Handbook of Computer Science and Engineering*. CRC Press, 2nd edition, 2004.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *ISOP*, pages 106–130, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.

[CCR14] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *SAS*, 2014.

[CCS13] Arlen Cox, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan. QUIC graphs: Relational invariant generation for containers. In *ECOOP*, pages 401–425, 2013.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *TACAS*, pages 93–107, 2013.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[Cha08] Bor-Yuh Evan Chang. *End-User Program Analysis.* PhD thesis, University of California, Berkeley, 2008.

[CHJ12] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *OOPSLA*, pages 587–606, 2012.

[CHR12] Nathaniel Charlton, Ben Horsfall, and Bernhard Reus. Crowfoot: A verifier for higher-order store programs. In *VMCAI*, pages 136–151, 2012.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.

[CRJ12] Ravi Chugh, Patrick Maxim Rondon, and Ranjit Jhala. Nested refinements: a logic for duck typing. In *POPL*, pages 231–244, 2012.

[DDA10] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, pages 246–266, 2010.

[DDA11] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *POPL*, pages 187–200, 2011.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, pages 230–241, 1994.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[dMB09] Leonardo Mendonça de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52, 2009.

[FAFH09] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael W. Hicks. Static type inference for ruby. In *SAC*, pages 1859–1866, 2009.

[FL10] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, pages 10–30, 2010.

[Fla97] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis.* PhD thesis, Rice University, May 1997.

[Flo67] Robert Floyd. Assigning meaning to programs. *AMS Symposia on Applied Mathematics*, 19:19–31, 1967.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[GMS12] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *POPL*, pages 31–44, 2012.

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.

[GNS13] Philippa Gardner, Daiva Naudziuniene, and Gareth Smith. JuS: Squeezing the sense out of JavaScript programs. In *JSTools*, 2013.

[GRS05] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP*, pages 126–150, 2010.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.

[HWCK14] Ben Hardekopf, Ben Wiedermann, Berkeley R. Churchill, and Vineeth Kashyap. Widening for control-flow. In *VMCAI*, pages 472–491, 2014.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL*, pages 14–26, 2001.

[JM07] Ranjit Jhala and Kenneth L. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.

[JMT09] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *SAS*, pages 238–255, 2009.

[JMT10] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *SAS*, pages 320–339, 2010.

[Kle43] Stephen Cole Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, Jan 1943.

[Kri11] Neelakantan R. Krishnawami. *Verifying Higher-Order Imperative Programs with HIgher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2011.

[KSW+13] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *DLS*, pages 17–26, 2013.

[Kun07] Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[KV09] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, pages 470–485, 2009.

[LKR05] Patrick Lam, Viktor Kuncak, and Martin C. Rinard. Hob: A tool for verifying data structure consistency. In *CC*, pages 237–241, 2005.

[LWJ+12] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL*, 2012.

[McC62] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[McM08] Kenneth L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, pages 413–427, 2008.

[MMLH+08] Mark Marron, Mario Méndez-Lojo, Manuel V. Hermenegildo, Darko Stefanovic, and Deepak Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, pages 43–49, 2008.

[MRV12] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. Modular heap analysis for higher-order programs. In *SAS*, pages 370–387, 2012.

[MSHK07] Mark Marron, Darko Stefanovic, Manuel V. Hermenegildo, and Deepak Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, pages 31–36, 2007.

[NTHH14] Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification. In *ICFP*, 2014.

[Pau07] L.D. Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, Feb 2007.

[Pnu81] M Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, pages 189–234, 1981.

[PTTC11] Tuan-Hung Pham, Minh-Thai Trinh, Anh-Hoang Truong, and Wei-Ngan Chin. Fixbag: A fixpoint calculator for quantified bag constraints. In *CAV*, pages 656–662, 2011.

[Pyt12] Python. Python 2.7.3 test suite, Apr 2012.

[RC11] Xavier Rival and Bor-Yuh Evan Chang. Calling context abstraction with shapes. In *POPL*, pages 173–186, 2011.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ml. In *POPL*, pages 77–88. ACM Press, 1989.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[RHS95] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[Riv05] Xavier Rival. *Traces Abstraction in Static Analysis and Program Transformation*. PhD thesis, École Normale Supérieure, 2005.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, pages 1–12, 2010.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[SAH+10] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[SBRY11] Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. Nested hoare triples and frame rules for higher-order store. *Logical Methods in Computer Science*, 7(3), 2011.

[SDC+12] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of JavaScript. In *ECOOP*, pages 435–458, 2012.

[SF92] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *LISP and Functional Programming*, pages 288–298, 1992.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[SKBG13] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *FSE*, 2013.

[SKK11] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.

[SPW09] Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. Abstraction refinement for quantified array assertions. In *SAS*, pages 3–18, 2009.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[ST07] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, pages 2–27, 2007.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[TF08] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL*, pages 395–406, 2008.

[Urb13] Caterina Urban. The abstract domain of segmented ranking functions. In *SAS*, pages 43–62, 2013.

[Ven14] Venafi. Venafi labs Q3 heartbleed threat research analysis, July 2014.

[VKSB14] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *DLS*, 2014.

[XJC09] Dana N. Xu, Simon L. Peyton Jones, and Koen Claessen. Static contract checking for haskell. In *POPL*, pages 41–52, 2009.

A. Full Example Analysis

In this chapter, I give the details and decisions that go into a full analysis based on the abstractions presented in this dissertation. This chapter uses an analysis like used by JSAna as explained in Chapter 6. To ensure that this example stays useful in explaining how the various abstractions work together, I have made some simplifications to eliminate extra indirections that are required in a JavaScript analysis. As a result, the structure of the analyzer for this example does not utilize the same simplification. This version is shown in Figure A.1.

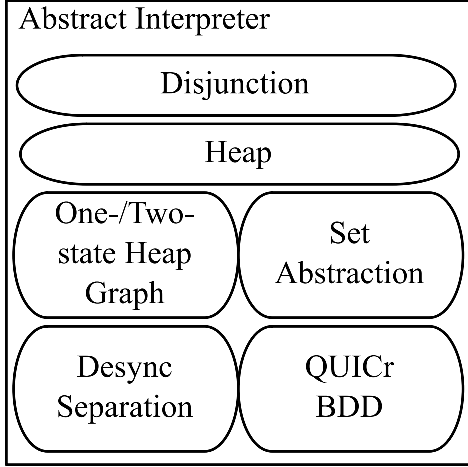


Figure A.1. – Architecture of the abstractions as used for the full example analysis

Because only the abstractions are being considered here, the program that is going to be analyzed is already lowered and simplified. The code that will be analyzed is shown in Figure A.2. The code here is an adapted version of `Class` that eliminates many of the complexities introduced along with JavaScript. Most significantly, it is assumed that a different call function that is more flexible can be used. It has lambda-calculus-style, pass-by-value semantics rather than JavaScript semantics. All other commands are presented as they would be analyzed and as given in the open-object-focused JavaScript.

The precondition for this function is given in Figure A.3. It consists of constraints on the three input variables. The `copy` variable points to a function value @1. This function value is the identifier of the `copy` function also given in Figure A.2. This precondition asserts that the copy variable is necessarily bound to the copy function. If this assumption is changed, then the precondition is not satisfied and any resulting inference cannot be relied upon. The `cfgc` variable points to a non-summary object at address a_1. It has an

```
function /* id:@1 */ copy(result, cfgc, exc) {
  for(var p in obj)
    /* point 2 */
  {
    if(p in exc) { } else {
      result[p] = cfgc[p];
    }
    /* point 3 */
  }
}

function constructor(copy,cfgc,args) {
  var init = cfgc.init;
  var result = {};
  var exc = {};
  exc.init = '';
  /* point 1 */
  copy(result, cfgc, exc);
  /* point 4 */
  init(result, args);
  /* point 5 */
  return result;
}
```

Figure A.2. – Adapted and simplified version of the `Class` function to be analyzed

initial set of attribute names F_1, an attribute/value tracker t_1, and an initial set of values V_1. This represents a mostly unknown object because there are is only one constraint on F_1, that it must have an 'init' attribute. Finally, the `args` variable points to an object at address $\{a_2\}$. This is a different object from $\{a_1\}$ because of the separation constraints implicit in these heap graphs. However, similar to $\{a_1\}$ is is an unknown object with attribute names F_2, attribute/value tracker t_2, and values V_2.

Figure A.3. – Precondition for analysis of `constructor` shown in Figure A.2

Up to program point 1, there are three simple statements. The first materializes the `init` function from the object pointed to by `cfgc`. This materialization splits F_1 into two partitions $F1_1$ and {'init'}, each of which gets a copy of the attribute/value tracker. This also materializes a value $\{V_1\}$ from V_1. Note that when values are materialized, it does not produce a partition like when attributes are materialized. This is because there is no separation requirement on values and thus it is not important to remove the materialized value from the set of values when it is materialized. Additionally, the `result`, and `exc` variables and corresponding objects were created. The object pointed to by `exc` is initialized with the attribute 'init'. The result is shown in Figure A.4

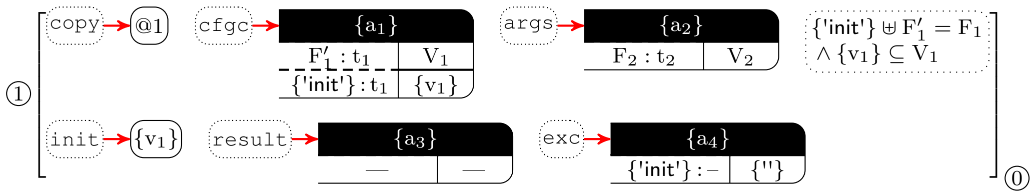


Figure A.4. – Analysis state at point 1 shown in Figure A.2

To reach program point 2, the function `copy` has to be called. To resolve this call, the analysis relies upon the given precondition for the analyzed function. This precondition indicates that `copy` points to function with id @1. Because the code for function id @1 is available, there is no need to desynchronize this function call. Instead the analysis (as it is today) does a context sensitive analysis. For simplicity, here, the parameters in the `copy` function are the same as the variables that are used. In general, a new environment is added that adds new variables that point to the objects that were passed as parameters.

In the copy function, an iteration runs. To perform this iteration, the attributes of the object that are being iterated over are partitioned into two: F_i, a variable that keeps track of the attributes still to visit, which is initially all of the attributes and F_o, a variable that keeps track of all of the already visited attributes, which is initially empty. The initial candidate invariant is shown in Figure A.5.

Within the loop, there is a case split that occurs. Either the attribute is 'init' or not. If it is not 'init', the attribute/value pair and the attribute/value tracker is copied from A_1 to A_3. If it is init, nothing happens. After completing this branch, they case split

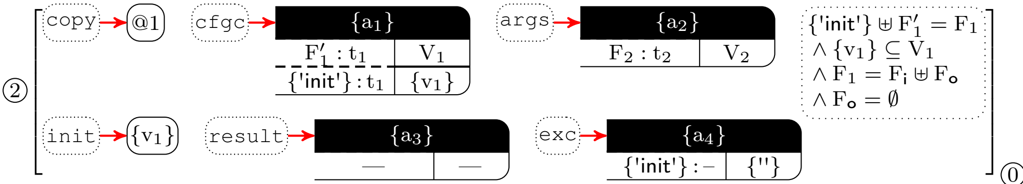


Figure A.5. – Analysis state at point 2, iteration 1 shown in Figure A.2

is maintained. Only immediately before the end of the loop need they be joined. At this point, the analysis produces Figure A.6, which shows the conditional addition of the attribute f to the result object.

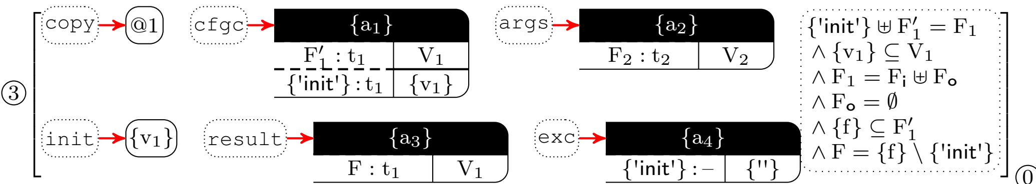


Figure A.6. – Analysis state at point 3, iteration 1 shown in Figure A.2

Coming back to the head of the loop, the analysis finds that the candidate loop invariant is insufficient and thus it performs a widening to infer a new candidate loop invariant. This widening drops the constraint that F_o is empty, which allows for a more general candidate loop invariant. This more general result is shown in Figure A.7.

This widening requires significant manipulation of the set abstraction. The set abstraction must determine that the abstraction in Figure A.5 and the end of the loop body imply that $F'_o = F_o \cap F'_1$.

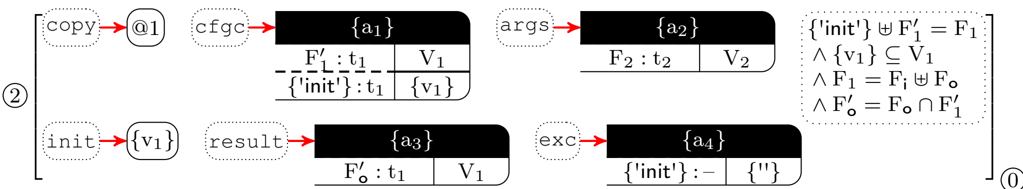


Figure A.7. – Analysis state at point 2, iteration 2 shown in Figure A.2

Program point 3 in the second iteration of the loop is shown in Figure A.8. The important change is that there is now a second partition in the result object a_3. One partition contains all attributes added by previous iterations, the other contains the attribute added this iteration, if any. Once again, because the two cases are joined, this addition is conditional upon whether the attribute is also in a_4.

The analysis checks to see if the state at the end of the loop body is contained in the candidate loop invariant. It is, so the candidate loop invariant is an actual loop invariant. Figure A.9 shows this loop invariant.

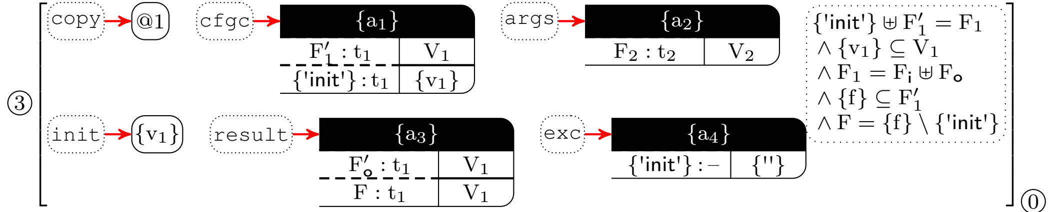


Figure A.8. – Analysis state at point 3, iteration 2 shown in Figure A.2

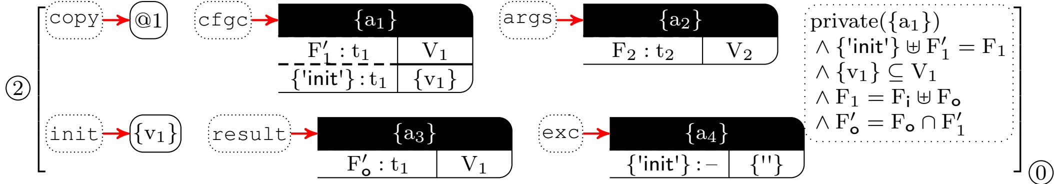


Figure A.9. – Analysis state at point 2, fixpoint shown in Figure A.2

Upon exiting the loop, the set F_o is equal to the set of attributes of object A_1. Therefore, F_o can be replaced with F_1, causing the sole partition of object A_3 to be F'_1. After exiting the function, the analysis returns to the previously used environment. This is shown in full in Figure A.10.

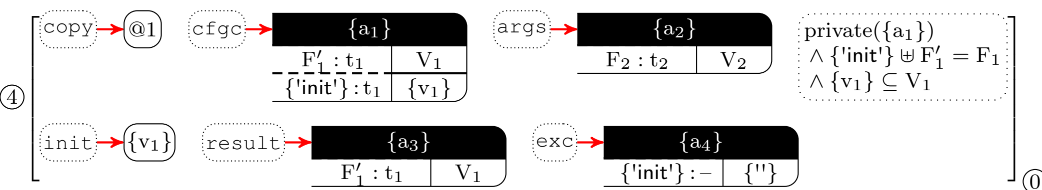


Figure A.10. – Analysis state at point 4 shown in Figure A.2

To proceed beyond program point 4, the analysis calls the function pointed to by `init`. Unlike the previous function call that could be resolved to @1, this call can only be resolved to a symbol V_1. Because the effect of this function is unknown, the analysis will use desynchronized separation of capture those effects. To introduce a desynchronized separation, it is necessary to first frame the function to separate the part of the heap that is assumed to be unmodified by the function call from the the part that may be modified by the function call. To do this, the analyzer uses a simple heuristic analysis. The heuristic analysis determines that anything that is reachable from the arguments of the called function may be modified by the function (more generally, it also includes anything reachable from the global object). This means that the only object that are not in the frame are A_2 and A_3, which are the arguments to the function. Desynchronized separation is introduced with these objects to get the analysis state at program point 5, which is shown in Figure A.11

As a result of the desynchronization, the analysis is able to compute that the result of

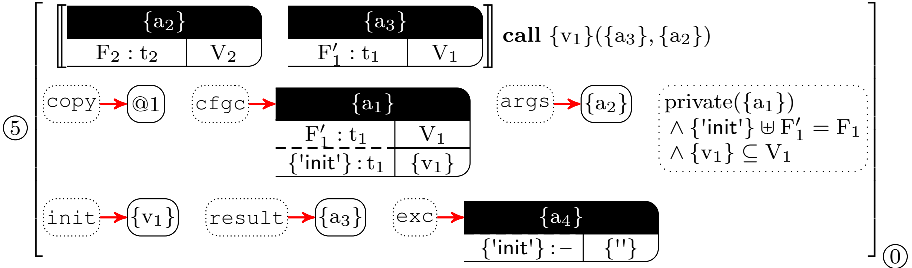


Figure A.11. – Analysis state at point 5 shown in Figure A.2

the function is A_4 even though A_4 itself is desynchronized. This is because `result` is not desynchronized and thus the value of `result` can be accessed. Since this value is a pointer, it is valid to return that pointer without knowledge of what it points to.

Because it is not desynchronized the analysis, the analysis proves (assuming a resynchronization is possible because the framing was sufficiently precise, which in this case it is) that object a_1 is unchanged by calling the constructor. This means that the class itself is not mutated by the call. Additionally, the analysis proves that the appropriate copy happened into the result object a_3 prior to running the `init` function. This represents the desired behavior of a class.

B. Detailed Proofs

B.1. HOO Materialization Soundness

Theorem 8 (Soundness of Materialization). *Restatement of Theorem 1.*
If $D_1 \Rightarrow D_2$, for all η, σ_1, σ_2, $(\eta, \sigma_1, \sigma_2) \in \gamma(D_1)$ implies that $(\eta, \sigma_1, \sigma_2) \in \gamma(D_2)$.

Proof. By induction on a derivation of $D_1 \Rightarrow D_2$. By inversion there are two cases:

Case Mat-Disj Assume an arbitrary η, σ_1, and σ_2 and that $(\eta, \sigma_1, \sigma_2) \in \gamma(D_1 \vee D_2)$. Show $(\eta, \sigma_1, \sigma_2) \in \gamma(D_1 \vee D_2')$. Unfolding the definition of γ gives either that $(\eta, \sigma_1, \sigma_2) \in \gamma(D_1)$ or $(\eta, \sigma_1, \sigma_2) \in \gamma(D_2)$.

Sub-Case $(\eta, \sigma_1, \sigma_2) \in \gamma(D_1)$ Unfolding the definition of γ gives the goal $(\eta, \sigma_1, \sigma_2) \in \gamma(D_1) \vee (\eta, \sigma_1, \sigma_2) \in \gamma(D_2')$. The first disjunct matches the case, completing this sub-case.

Sub-Case $(\eta, \sigma_1, \sigma_2) \in \gamma(D_2)$ Unfolding the definition of γ gives the goal $(\eta, \sigma_1, \sigma_2) \in \gamma(D_1) \vee (\eta, \sigma_1, \sigma_2) \in \gamma(D_2')$. By the I.H. $(\eta, \sigma_1, \sigma_2) \in \gamma(D_2)$ implies that $(\eta, \sigma_1, \sigma_2) \in \gamma(D_2')$, which matches the second disjunct, completing this sub-case and this case.

Case Mat-Heap Assume an arbitrary η, σ_1, and σ_2 and that $(\eta, \sigma_1, \sigma_2) \in \gamma([H_2]_{H_1} \mathrel{\vdots} P)$. From this there exists a tracker map μ such that $(\eta, \mu, \sigma_1) \in \gamma(H_1)$ and $(\eta, \mu, \sigma_2) \in \gamma(H_2)$ and $\eta \in \gamma(P)$.

Show that $(\eta, \sigma_1, \sigma_2) \in \gamma\left(\bigvee \left\{ [H_i]_{H_1} \mathrel{\vdots} P_i \mid H_i \mathrel{\vdots} P_i \in \bar{H} \right\}\right)$. Or by unfolding γ, show that there exists an $(H_i \mathrel{\vdots} P_i) \in \bar{H}$ such that $(\eta, \mu, \sigma_1) \in \gamma(H_1)$ and $(\eta, \mu, \sigma_2) \in \gamma(H_i)$ and $\eta \in \gamma(P_i)$.

Because $(\eta, \mu, \sigma_1) \in \gamma(H_1)$ does not depend on any of the existentially quantified variables, it can be lifted outside the existential and follows immediately from the hypothesis. The remaining existential follows trivially from Lemma 1, completing this case and the proof. ☐

Lemma 1 (Soundness of Heap Materialization). *If $H \mathrel{\vdots} P \Rightarrow \bar{H}$, for all η, μ, σ, $(\eta, \mu, \sigma) \in \gamma(H)$, $\eta \in \gamma(P)$, there exists a $(H' \mathrel{\vdots} P') \in \bar{H}$ such that there exists a η' such that $(\eta, \mu, \sigma) \in \gamma(H')$ and $\eta' \in \gamma(P')$ and $\eta \subseteq \eta'$.*

Proof. By induction on the derivation of $H \mathrel{\vdots} P \Rightarrow \bar{H}$. By inversion there are three cases:

Case Mat-Heap-Frame This case is a standard separation logic frame rule. It relies upon the fact that any new symbols introduced are fresh and thus do not interfere with existing symbols. See **(author?)** [Rey02].

Case Mat-Addr The assumptions for the MAT-ADDR rule take indicate that $\gamma(P)$ is being partitioned. By Lemma 3, $\gamma(P')$ and $\gamma(P'')$ are those partitions that correspond respectively to the two result cases: $\gamma(\{\mathrm{a}\} \cdot \langle O \rangle * \mathrm{A}' \cdot \langle O \rangle \mathbin{\vdots} P')$ and $\gamma(\mathrm{A} \cdot \langle O \rangle \mathbin{\vdots} P'')$. Because $\{\mathrm{a}\} \uplus \mathrm{A}' = \mathrm{A}$ from partitioning, the first case trivially holds. Similarly, the second case is implied directly by the hypothesis.

Case Mat-Obj By Lemma 2 there exists at least one element in $\bar{O}$ such that □

Lemma 2 (Soundness of Object Materialization). *If $O \mathbin{\vdots} P \Rightarrow \bar{O}$, for all η, μ, o, d, $(\eta, \mu, o, d) \in \gamma(O)$, $\eta \in \gamma(P)$, there exists a $(O' \mathbin{\vdots} P') \in \bar{O}$ such that there exists a η' such that $(\eta, \mu, o, d) \in \gamma(O')$ and $\eta' \in \gamma(P')$ and $\eta \subseteq \eta'$.*

Proof. By induction on the derivation of $O \mathbin{\vdots} P \Rightarrow \bar{O}$. By inversion there are three cases:

Case Mat-Obj-Frame Like the MAT-HEAP-FRAME rules, this follows from standard separation logic [Rey02]. This is, in effect, a nested heap within an object, where the heap maps attributes to values (that include attributes).

Case Mat-Attr Like the MAT-ADDR case above, this follows directly from Lemma 3. The partitioning establishes the two result cases.

Case Mat-Value In $\gamma(\{\mathrm{f}\} : \mathrm{t} \mapsto \mathrm{V} \mathbin{\vdots} P)$, there can only be one and only one element of V that is mapped to by f. The introduction of a fresh v specifies which element of V that is mapped to.

Lemma 3 (Partitioning). *For any pure domain instance P, and for any symbols* f *and* F*, there exists an* F′ *such that*

$$P \Rightarrow \big(P \wedge \{\mathrm{f}\} \uplus \mathrm{F}' = \mathrm{F}\big) \vee (P \wedge \{\mathrm{f}\} \cap \mathrm{F} = \emptyset)$$

where F′ *is fresh.*

Proof. By rules of distribution get that

$$P \Rightarrow P\,\big(\{\mathrm{f}\} \uplus \mathrm{F}' = \mathrm{F} \vee \{\mathrm{f}\} \cap \mathrm{F} = \emptyset\big)$$

Prove $(\{\mathrm{f}\} \uplus \mathrm{F}' = \mathrm{F} \vee \{\mathrm{f}\} \cap \mathrm{F} = \emptyset)$ is a tautology. Do a case split on $\{\mathrm{f}\} \cap \mathrm{F} = \emptyset$:

Case $\{\mathrm{f}\} \cap \mathrm{F} = \emptyset$ Reflexivity with second disjunct.

Case $\{\mathrm{f}\} \cap \mathrm{F} \neq \emptyset$ Trivially, $\{\mathrm{f}\} \subseteq \mathrm{F}$. Choose $\mathrm{F}' = \mathrm{F} \setminus \{\mathrm{f}\}$. From this get $\{\mathrm{f}\} \cup \mathrm{F}' = \mathrm{F}$ and $\mathrm{F}' \cap \{\mathrm{f}\} = \emptyset$. Therefore $\{\mathrm{f}\} \uplus \mathrm{F}' = \mathrm{F}$, which is reflexive with the first disjunct. □

B.2. HOO Transfer Function Soundness

Theorem 9 (Soundness of Transfer Functions). *Restatement of Theorem 2. Transfer functions are sound because:*

$$\forall k, \sigma_0, \sigma, \sigma', D, D'.\ \langle\sigma\rangle\, k\, \langle\sigma'\rangle \ \textit{and}\ [D]\ k\ [D'] \ \textit{and}\ \exists\eta.\ (\eta, \sigma_0, \sigma) \in \gamma(D) \\ \textit{implies}\ \exists\eta'.\ (\eta', \sigma_0, \sigma') \in \gamma(D')$$

Proof. By induction on the derivation of $\langle\sigma\rangle\, k\, \langle\sigma'\rangle$ and by induction on the derivation of $[D]\ k\ [D']$. Consider a case for each k:

Case $k =$ `x = y[z]` By inversion consider follow rules for concrete evaluation and abstract evaluation:

Sub-Case C-Ld-P and T-Read-P From the hypothesis we have

$$(\eta, \sigma) \in \gamma(\mathrm{x} \mapsto \mathrm{V_x} * \mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle\{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O\rangle \mathbin{¦} P)$$

so by the rules of $*$, this is equivalent to:

$$\begin{aligned} &(\eta, \sigma_x) \in \gamma(\mathrm{x} \mapsto \mathrm{V_x}) \\ &\wedge (\eta, \sigma_r) \in \gamma(\mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle\{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O\rangle) \\ &\wedge \eta \in \gamma(P) \\ &\wedge \sigma = \sigma_x \uplus \sigma_r \end{aligned}$$

Applying a similar option to the goal of:

$$(\eta', \sigma') \in \gamma(\mathrm{x} \mapsto \{\mathrm{v}\} * \mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle\{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O\rangle \mathbin{¦} P)$$

gives a new goal of (choosing $\eta' = \eta$):

$$\begin{aligned} &(\eta, \sigma'_x) \in \gamma(\mathrm{x} \mapsto \{\mathrm{v}\}) \\ &\wedge (\eta, \sigma'_r) \in \gamma(\mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle\{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O\rangle) \\ &\wedge \eta \in \gamma(P) \\ &\wedge \sigma' = \sigma'_x \uplus \sigma'_r \end{aligned}$$

From which the $\eta \in \gamma(P)$ can be immediately dispatched. By Lemma 4, we know that $\sigma_r = \sigma'_r$ and therefore, we have the following hypothesis:

$$\begin{aligned} &(\eta, \sigma_x) \in \gamma(\mathrm{x} \mapsto \mathrm{V_x}) \\ &\wedge (\eta, \sigma'_r) \in \gamma(\mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle\{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O\rangle) \\ &\wedge \eta \in \gamma(P) \\ &\wedge \sigma = \sigma_x \uplus \sigma'_r \end{aligned}$$

which allows the $(\eta, \sigma'_r) \in \gamma(\mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle\{\mathrm{f}\} : \mathrm{t} \mapsto \{\mathrm{v}\}; O\rangle)$ portion of the goal to be dispatched, which leaves the following two parts of the goal to prove:

$$\begin{aligned} &(\eta, \sigma'_x) \in \gamma(\mathrm{x} \mapsto \{\mathrm{v}\}) \\ &\wedge \sigma' = \sigma'_x \uplus \sigma'_r \end{aligned}$$

Again, by Lemma 4, we know that $\sigma'_x = \sigma_x$ where x's value has been replaced by v. Since this is the only portion of σ_x, σ'_x is therefore in the concretization of $\gamma(\mathrm{x} \mapsto \{\mathrm{v}\})$ and similarly $\sigma' = \sigma'_x \uplus \sigma'_r$, completing this sub-case.

Sub-Case C-Ld-N and T-Read-N From the hypothesis, we have:

$$(\eta, \sigma) \in \gamma(\mathrm{x} \mapsto \mathrm{V_x} * \mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle O \rangle \,\text{¦}\, P)$$

which can be rewritten as:

$$\begin{aligned} &(\eta, \sigma_x) \in \gamma(\mathrm{x} \mapsto \mathrm{V_x}) \\ &\wedge (\eta, \sigma_r) \in \gamma(\mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle O \rangle) \\ &\wedge \eta \in \gamma(P) \\ &\wedge \sigma = \sigma_x \uplus \sigma_r \end{aligned}$$

The same rewrite can be performed on the goal giving:

$$\begin{aligned} &(\eta', \sigma'_x) \in \gamma(\mathrm{x} \mapsto \{\mathsf{undef}\}) \\ &\wedge (\eta', \sigma'_r) \in \gamma(\mathrm{y} \mapsto \{\mathrm{a}\} * \mathrm{z} \mapsto \{\mathrm{f}\} * \{\mathrm{a}\} \cdot \langle O \rangle) \\ &\wedge \eta' \in \gamma(P) \\ &\wedge \sigma' = \sigma'_x \uplus \sigma'_r \end{aligned} \qquad \square$$

Similar to above, if we can prove the first conjunct, the remaining follow trivially. The first conjunct is proven simply by the concrete evaluation, which dictates the value corresponding to x is undef.

Other Sub-Cases These trivially do not apply. The abstract evaluation that assumes the attribute is present cannot be matched to the concrete evaluation that assumes the attribute is not present. Similar for other cases.

Case $k =$ **`x[y] = z`** By inversion, the rule C-St applies for concrete and T-Write applies for the abstract. Most of this case proceeds as the above two subcases. The portion of the abstract heap $\mathrm{x} \mapsto \{\mathrm{a}\} * \mathrm{y} \mapsto \{\mathrm{f}\} * \mathrm{z} \mapsto \{\mathrm{v}\}$ is trivially maintained in the concrete. The portion that represents the object is replaced in the concrete (via Lemma 4) with a new object that has the attribute/value replaced with p and v respectively. In the abstract, the same operation applies. Via Lemma 5, the object is replaced with the same object with the abstract values that correspond to p and v.

Other Cases Other cases are either similar to the above two cases or are standard control flow operations and follow abstract interpretation for imperative languages.

Lemma 4 (Lookup Array). *For any a, b, r, and r', $r' = (a, b) :: r$ implies that* $\text{Lookup}\,(a, r') = b$ *and for any a' and b', if $a \neq a'$,* $\text{Lookup}\,(a', r') = b'$ *if and only if* $\text{Lookup}\,(a', r) = b'$.

Proof. This is a simple instantiation of McCarthy's array axioms [McC62]. □

Lemma 5 (Object Overwrite). *For all η, o, d, O, O', P, and* f, *assuming $(\eta, o, d) \in \gamma(O)$, $(\eta', o', d') \in \gamma(O')$, and $O \setminus \{\mathrm{f}\} \mathbin{\vdots} P \Rightarrow O' \mathbin{\vdots} P'$, implies that $\eta\{\mathrm{f}\} \notin d'$ and for all $p \in d$, v, $d(p) = v$ and $p \neq \mathrm{f}$ implies $d'(p) = v$.*

Proof. By induction over the derivation of $O \setminus \{\mathrm{f}\} \mathbin{\vdots} P \Rightarrow O' \mathbin{\vdots} P'$. □

B.3. HOO Join Soundness

Theorem 10 (Join Soundness). *Restatement of Theorem 3.*
Join is sound under matchings M_1, M_2, P_J because

$$
\begin{array}{l}
\textit{if } P \vdash [H_1]_{H_0} \mathbin{\vdots} P_1 \sqcup [H_2]_{H_0} \mathbin{\vdots} P_2 \leadsto [H_3]_{H_0} \mathbin{\vdots} P_3 \textit{ then} \\
\quad \forall \sigma_0, \sigma, \eta_1, \eta_2.\ (\eta_1, \sigma_0, \sigma) \in \gamma([H_1]_{H_0} \mathbin{\vdots} P_1) \vee (\eta_2, \sigma_0, \sigma) \in \gamma([H_2]_{H_0} \mathbin{\vdots} P_2) \\
\qquad \wedge \forall (\bar{\mathrm{V}}_1, \bar{\mathrm{V}}_2, \mathrm{V}_3) \in P.\ \bigcup \{\, \eta_1(\mathrm{V}_1) \mid \mathrm{V}_1 \in \bar{\mathrm{V}}_1 \,\} = \bigcup \{\, \eta_2(\mathrm{V}_2) \mid \mathrm{V}_2 \in \bar{\mathrm{V}}_2 \,\} \Rightarrow \\
\qquad\quad \exists \eta_3.\ (\eta_3, \sigma_0, \sigma) \in \gamma([H_3]_{H_0} \mathbin{\vdots} P_3) \\
\qquad\qquad \wedge \forall (\bar{\mathrm{V}}_1, \bar{\mathrm{V}}_2, \mathrm{V}_3) \in P.\ \bigcup \{\, \eta_1(\mathrm{V}_1) \mid \mathrm{V}_1 \in \bar{\mathrm{V}}_1 \,\} = \eta_3(\mathrm{V}_3)
\end{array}
$$

where P is a uniform, combined version of M_1, M_2, and P_J and is defined as

$$
P \stackrel{def}{=} \left\{ (\bar{\mathrm{V}}_1, \bar{\mathrm{V}}_2, \mathrm{V}_3) \;\middle|\; \begin{array}{l} \mathrm{V}_3 \in \mathrm{Codom}\,(M_1) \cup \mathrm{Codom}\,(M_2) \\ \wedge\, \bar{\mathrm{V}}_1 = \{\, \mathrm{V}_1 \mid M_1(\mathrm{V}_1) = \mathrm{V}_3 \,\} \\ \wedge\, \bar{\mathrm{V}}_2 = \{\, \mathrm{V}_2 \mid M_1(\mathrm{V}_2) = \mathrm{V}_3 \,\} \end{array} \right\} \cup P_J
$$

Proof. The purpose of P is to unify the three different matchings M_1, M_2, and P_J. The purpose of these three mappings is to enable algorithm inference of the mappings. Once they are inferred, they say the same thing. This is what P represents. Each element $(\bar{\mathrm{V}}_1, \bar{\mathrm{V}}_2, \mathrm{V}_3)$ maps the union of the symbols in $\bar{\mathrm{V}}_1$, which is a set of symbols, and the union of the symbols in $\bar{\mathrm{V}}_2$, which is also a set of symbols, to a symbol in the join result V_3. This is unification is responsible for the complexity of the theorem. In short the theorem states that if there is a σ in the concretization of $H_1 \mathbin{\vdots} P_1$ or in $H_2 \mathbin{\vdots} P_2$, then σ is in the join result $H_3 \mathbin{\vdots} P_3$.

By induction on the derivation of join, and by inversion there are two join rules from Table 4.1 that can be applied to partial objects.

Case Empty Objects This case is trivial. The two input objects are empty and the result object is empty. The concretizations are identical.

Case Matched Attribute Sets The unified matching P dictates which partitions shall be merged. By the induction hypothesis, the concretization of the merged partitions from $H_1 \mathbin{¦} P_1$ and $H_2 \mathbin{¦} P_2$ are identical. In the join result, the resulting single partition V_3, constrained by P_3 is selected to have an identical concretization to the merged partitions. Thus the joining partitions is sound.

The join of objects is similar. Because the matching pushes merged objects into the set abstractions P_1, P_2, and P_3, any matching produces a valid join. □

B.4. HOO Inclusion Soundness

Theorem 11 (Inclusion Soundness). *Restatement of Theorem 4.*
Inclusion checking is sound under matchings M, P_I because assuming that P is defined as follows:

$$P \stackrel{def}{=} \left\{ (\bar{\mathrm{V}}_a, \mathrm{V}_b) \;\middle|\; \mathrm{V}_b \in \mathrm{Codom}\,(M) \wedge \bar{\mathrm{V}}_a = \{ \mathrm{V}_a \mid M(\mathrm{V}_a) = \mathrm{V}_b \} \right\} \cup P_I$$

If $M, P_I \vdash H_a, P_a \sqsubseteq H_b, P_b$ then

$$\forall \eta_a, \sigma_0, \sigma.\ (\eta_a, \sigma_0, \sigma) \in \gamma([H_a]_{H_0} \mathbin{¦} P_a) \Rightarrow$$
$$\exists \eta_b.\ (\eta_b, \sigma_0, \sigma) \in \gamma([H_b]_{H_0} \mathbin{¦} P_b) \wedge \forall (\bar{\mathrm{V}}_a, \mathrm{V}_b) \in P.\ \bigcup \left\{ \eta_a(\mathrm{V}_a) \;\middle|\; \mathrm{V}_a \in \bar{\mathrm{V}}_a \right\} = \eta_b(\mathrm{V}_b)$$

Proof. This follows the same pattern as join soundness above. Once again P serves as a unification between the object mapping M and the partition mapping P_I. The inclusion version of the above rules are applied in the same way. □

B.5. Desynchronization Introduction Soundness

Theorem 12 (Soundness of Desynchronization Introduction). *Restatement of Theorem 5.*
The desynchronization introduction transfer function is sound because

$$\forall k, \sigma, \sigma', D, D'.\ \langle \sigma \rangle\, k \left\langle \sigma' \right\rangle \textit{ and } [D]\ k\ \left[D'\right] \textit{ and } \exists \eta.\ (\eta, \sigma) \in \gamma(D)$$
$$\textit{implies } \exists \eta'.\ (\eta', \sigma') \in \gamma(D')$$

Proof. The only k of concern for desynchronization is the function call. By inversion the Desync-Intro rule can apply. Here this case is considered.

The Desync-Intro rule evaluates reach(), which effectively splits the abstract heap, but correspondingly the concrete heap into two parts $\sigma = \sigma_u \uplus \sigma_r$. The σ_u part is trivially in the concretization of the resulting heap. The remaining σ_r' of the resulting heap is derived from σ_r by the concretization itself, which has an embedded concrete transition of $\langle \sigma_r \rangle\, k \, \langle \sigma_r' \rangle$.

B.6. QUIC Graphs Inference Soundness

Theorem 13 (Inference Soundness). *Restatement of Theorem 6.*
Inference is sound because the following two conditions hold:

1. *if* $(G, B) \vdash \dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_{B_e}$*, then* $\gamma((G, B)) \subseteq \gamma\left(\dot{\bigcap} \bar{T}^i \dot{\subseteq} \dot{\bigcup} \bar{T}^u \Big|_{B_e}\right)$.
2. *if* $(G, B) \vdash \mathrm{f}_1 = \mathrm{f}_2$*, then for all* $(\eta, \eta_B) \in \gamma((G, B))$*, it is that* $\eta_B(\mathrm{f}_1) = \eta_B(\mathrm{f}_2)$.

Proof. For clarity, this proof is presented using the logical correspondence. From the logical perspective, this theorem shows that starting with a $G \wedge B$, applying a rule yields a $G' \wedge B'$ that should be logically equivalent.

Case Emp For any T, $\emptyset \subseteq T$, so this is trivially true.

Case Self-Loop For any T, $T \subseteq T$, so this is trivially true.

Case Self-Prop If $T \subseteq \{ \nu \in T \mid B_b[\nu] \}$, for all $\nu \in T$, $B_b[\nu]$ holds. Also, If $T \subseteq \left\{ \nu \in \bigcup \bar{T} \;\middle|\; B_a[\nu] \right\}$, then for all $\nu \in T$, $B_a[\nu]$. Consequently, for all $\nu \in T$, $B_a[\nu]$ and $B_b[\nu]$ holds.

Case Add-Left If an element ν that is in each of $\bar{T}^i$ is also in one of $\bar{T}^u$, considering fewer elements by adding an additional T to $\bar{T}^i$ does not restrict the contents of any $\bar{T}^u$.

Case Add-Right If an element ν that is in each of $\bar{T}^i$ is also in one of $\bar{T}^u$, it is also in one of $\bar{T}^u, T$.

Case Union-Prop If an element $\nu \in T^i$ and meeting $B_0[\nu]$ is therefore in at least one of $T_1^i, \ldots, T_n^i$ where each $\nu_1 \in T_1$ meets $B_1[\nu_1]$, …, $\nu_n \in T_n$ meets $B_n[\nu_n]$, that element ν is also in the disjunction of $B_0[\nu] \wedge (B_1[\nu] \vee \ldots \vee B_n[\nu])$.

Case Inter-Prop Trivial. Like Union-Prop.

Case Union-Trans This rule states the following:

$$\left(\forall \nu. \bigwedge_{T_1 \in \bar{T}_a^i} \nu \in T_1 \to \nu \in T \vee \bigvee_{T_2 \in \bar{T}_a^u} \nu \in T_2\right)$$
$$\wedge \left(\forall \nu. \nu \in T \wedge \bigwedge_{T_1 \in \bar{T}_b^i} \nu \in T_1 \to \bigvee_{T_2 \in \bar{T}_b^u} \nu \in T_2\right)$$

implies that

$$\forall \nu. \bigwedge_{T_1 \in \bar{T}_a^i} \nu \in T_1 \wedge \bigwedge_{T_2 \in \bar{T}_b^i} \nu \in T_2 \rightarrow \bigvee_{T_3 \in \bar{T}_a^u} \nu \in T_3 \vee \bigvee_{T_4 \in \bar{T}_b^u} \nu \in T_4 \qquad \square$$

Fix ν and introduce $\bigwedge_{T_1 \in \bar{T}_a^i} \nu \in T_1$ and $\bigwedge_{T_2 \in \bar{T}_b^i} \nu \in T_2$. By the first conjunct in the hypothesis, $\nu \in T \vee \bigvee_{T_2 \in \bar{T}_a^u} \nu \in T_2$.

Case $\nu \in T$: from second conjunct and introduced terms, get that $\nu \in \bigvee_{T_2 \in \bar{T}_b^u} \nu \in T_2$, which matches the second disjunct.

Case $\nu \in \bigvee_{T_2 \in \bar{T}_a^u} \nu \in T_2$: matches first disjunct.

Base constraints follow the above.

Case Inter-Trans Like UNION-TRANS.

Case Base-Str Because ν does not occur in B, adding B to each comprehension just repeats the external constraint that already exists.

Case Eq-Base For any $\mathrm{f}_1 = \mathrm{f}_2$ this trivially implies that $\{\mathrm{f}_1\} = \{\mathrm{f}_2\}$.

Case Eq-Set Same as EQ-BASE.

Case Double-Edge Trivial: $T_1 \subseteq \{ \nu \in T_2 \mid B_a[\nu] \} \wedge T_1 \subseteq \{ \nu \mid T_2 | B_b[\nu] \}$ implies that $T_1 \subseteq \{ \nu \in T_2 \mid B_a[\nu] \wedge B_b[\nu] \}$

B.7. QUIC Graphs Complexity

Theorem 14 (Inference Complexity). *Restatement of Theorem 7.*
There are $O(2^n)$ possible hyperedges in a QUIC graph with n vertices.

Proof. Each edge starts from a subset of n vertices and each edge ends at a subset of n vertices. There are therefore 2^n sources for each edge and 2^n destinations for each edge. Therefore, there are 2^{2n} possible edges in a QUIC graph with n vertices. $\square$

C. Inclusion Algorithm

Inclusion checking is similar to join in its specification and algorithm. Figure C.1 gives the rules for checking inclusion of abstract states. The LE-DISJ rule allows multiple disjuncts to be included in a single disjunct. Whereas, the LE-DISJ-DIR rule directly matches disjuncts, assuming that disjuncts can be freely rearranged. The main rule is LE-HEAP which checks inclusion of two heaps by matching roots and setting an initial mapping based on those roots, which can guide the rest of the matching process.

The actual matching of heap elements takes place in two rules: LE-EMP and LE-OBJ, which match empty heaps and partial heaps respectively. Once a portion of a heap is matched, the remaining parts are matched extending the matchings. The LE-OBJ rule is responsible for delegating to the table of object matching rules like join uses. These object matching rules produce new matchings as a result and those resulting matchings are used for the remainder of the heap.

The inclusion checking templates work like their join-templates counterparts. The first template matches a single partition with a single partition. The second template matches some number of partitions in the first object with a single partition in the second object. When this occurs, those partitions are not considered in future template applications. The application of templates is repeated until no partitions remain in the matched objects.

The algorithm for inclusion checking works the same way as the join algorithm. It uses allocation site information to determine possible valid matchings. These are used to seed the initial matching:

Table C.1. – Inclusion checking templates match objects in two abstract heaps. Matchings M, P_I are generated on the fly and used in the set domain join after the heaps are compared.

Prerequisites	H_1, P_1	$\sqsubseteq$	H_2, P_2	$\rightsquigarrow$	Result
$M(A_1) = A_2$	A_1 F_1 | V'_1	$\sqsubseteq$	A_2 F_2 | V'_2	$\rightsquigarrow$	$M(V'_1) = V'_2$, $(\{F_1\}, F_2) \in P_I$
$M_1(A_1) = A_3$ $M_2(A_2) = A_3$ remainder of object matches	A_1 ⋮ | ⋮ F_1^i | V_1^i ⋮ | ⋮ F_1^m | V_1^m ⋮ | ⋮	$\sqsubseteq$	A_2 ⋮ | ⋮ F_2 | V_2 ⋮ | ⋮	$\rightsquigarrow$	$(\{F_1^i, \cdots, F_1^m\}, F_2) \in P_I$ $M(V_1^i) = V_2$, ⋮ $M(V_1^m) = V_2$,

$$\boxed{D_1 \sqsubseteq D_2}$$

$$\textsc{Le-Disj}\ \frac{D_1 \sqsubseteq D_3 \qquad D_2 \sqsubseteq D_3}{D_1 \vee D_2 \sqsubseteq D_3} \qquad \textsc{Le-Disj-Dir}\ \frac{D_1 \sqsubseteq D_3 \qquad D_2 \sqsubseteq D_4}{D_1 \vee D_2 \sqsubseteq D_3 \vee D_4}$$

$$\textsc{Le-Heap}\ \frac{M_I, \{\} \vdash H_1 \,\dot{|}\, P_1 \sqsubseteq H_2 \,\dot{|}\, P_2 \qquad M_I = \bigcup_{\mathrm{A}_1 \in \mathrm{roots}(H_1) \cap \mathrm{roots}(H_2)} [\mathrm{A}_1 \mapsto \mathrm{A}_1]}{[H_1]_{H_0} \,\dot{|}\, P_1 \sqsubseteq [H_2]_{H_0} \,\dot{|}\, P_2}$$

$$\boxed{M, P_I \vdash H_1 \,\dot{|}\, P_1 \sqsubseteq H_2 \,\dot{|}\, P_2}$$

$$\textsc{Le-Emp}\ \frac{M, P_I \vdash P_1 \sqsubseteq P_2}{M, P_I \vdash \textsc{Emp} \,\dot{|}\, P_1 \sqsubseteq \textsc{Emp} \,\dot{|}\, P_2}$$

$$\textsc{Le-Obj}\ \frac{\begin{array}{c} M' = [\mathrm{A}_1 \mapsto \mathrm{A}_2] \cup M \\ M', P_I \vdash O_1, P_1 \sqsubseteq O_2, P_2 \rightsquigarrow M'', P'_I \qquad M'', P'_I \vdash H_1 \,\dot{|}\, P_1 \sqsubseteq H_2 \,\dot{|}\, P_2 \end{array}}{M_1, P_I \vdash \mathrm{A}_1 \cdot \langle O_1 \rangle \,*\, H_1 \,\dot{|}\, P_1 \sqsubseteq \mathrm{A}_2 \cdot \langle O_2 \rangle \,*\, H_2 \,\dot{|}\, P_2}$$

Figure C.1. – Rules for checking inclusion of two heaps. The Le-Disj rule permits checking inclusion of disjunction. The Le-Obj rule applies the templates given in Table C.1 to check inclusion of two objects.

$$M = \bigcup_{\mathrm{A} \in \mathrm{Dom}(H_1)} [\mathrm{A} \mapsto \mathrm{symbol}\left(\mathrm{alloc\text{-}id}\left(\mathrm{A}\right)\right)]$$

Under these assumptions, a matching is produced. If it passes the inclusion check for the set domain once there is no heap left to match, the inclusion check succeeds.

D. Detailed Tests for Single-State HOO and TAJS

This section shows the code evaluated in each test and the pre-condition used to initialize the state. Due to differences between TAJS and HOO, the code given here is simply the main loop, rather than the whole program. Each system requires special initialization. Additionally, the HOO implementation only accepts abstract syntax as input, so the AST was translated to abstract syntax by hand for these tests.

The properties that follow each test are checked against the inferred post-condition. These results are extracted through a combination of manual inspection of post-conditions and through test code. This extraction/test code is not shown. A ✓ is shown to indicate a property that can be proven given the post-condition and a ✗ is shown to indicate a property that cannot be proven given the post-condition.

There is an implicit assumption that `hasOwnProperty` is actually looked up in the global space and is correctly resolved. In absence of surrounding code, it is impossible to know that this functionality has not been shadowed. The assumption that this has not occurred is implicit in all of the preconditions.

Listing D.1 – Static

[EMP]

```
s = {
  x:"a";
  y:"b";
};
r = {};
for(var p in s) {
  if(s.hasOwnProperty(p))
    r[p] = s[p];
}
```

Property	TAJS	HOO
`r` $\neq$ `s`	✓	✓
`r["x"]` = `s["x"]`	✓	✓
`r["y"]` = `s["y"]`	✓	✓
$p \neq$ 'x' $\wedge\, p \neq$ 'y' $\rightarrow$ `r[`p`]` = `s[`p`]` = *undefined*	✓	✓

Listing D.2 – Copy

$[\mathtt{s} \mapsto \{a_1\} * \{a_1\} \cdot \langle F \mapsto A_2 \rangle]$

```
r = {};
for(var p in s) {
  if(s.hasOwnProperty(p))
    r[p] = s[p];
}
```

Property	TAJS	HOO
$\mathtt{r} \neq \mathtt{s}$	✓	✓
$p \in Fr \rightarrow p \in Fs$	✗	✓
$p \in Fs \rightarrow p \in Fr$	✗	✓
$p \in Fr \cap Fs \rightarrow \mathtt{r}[p] \in A_2$	✗	✓

Listing D.3 – Filter

$$\left[\begin{array}{l} \mathtt{r} \mapsto \{a_1\} * \{a_1\} \cdot \langle F_1 \mapsto V_1 \rangle * \mathtt{s} \mapsto \{a_2\} \\ * \{a_2\} \cdot \langle F_2 \mapsto V_2 \rangle * \mathtt{c} \mapsto \{a_3\} * \{a_3\} \cdot \langle F_3 \mapsto V_3 \rangle \end{array}\right]$$

```
for(var p in s) {
  if(s.hasOwnProperty(p)) {
    if(c.hasOwnProperty(p))
      r[p] = "conflict";
    else
      r[p] = s[p];
  }
}
```

Property	TAJS	HOO
$p \in Fs \rightarrow p \in Fr$	✗	✓
$p \in Fr \rightarrow p \in Fs \cup F_1$	✗	✓
$p \in Fr \wedge p \notin Fs \rightarrow p \in F_1$	✗	✓
$p \in Fr \wedge p \in Fs \wedge p \in Fc \rightarrow \mathtt{r}[p] =$ 'conflict'	✗	✓
$p \in Fr \wedge p \in Fs \wedge p \notin Fc \rightarrow \mathtt{r}[p] \in V_2$	✗	✓
$p \in Fr \wedge p \notin Fs \rightarrow \mathtt{r}[p] \in V_1$	✗	✓

Listing D.4 – Compose

$$\left[\begin{array}{l} \texttt{r} \mapsto \{a_1\} * \{a_1\} \cdot \langle F_1 \mapsto V_1 \rangle \\ * \ \texttt{a} \mapsto \{a_2\} * \{a_2\} \cdot \langle F_2 \mapsto V_2 \rangle \\ * \ \texttt{b} \mapsto \{a_3\} * \{a_3\} \cdot \langle F_3 \mapsto V_3 \rangle \\ * \ \texttt{c} \mapsto \{v_4\} \\ * \ \texttt{s} \mapsto \{a_4\} * \{a_4\} \cdot \langle F_4 \mapsto v_4 \rangle \\ \wedge F_4 = F_2 \cup F_3 \end{array}\right]$$

```
for(p in s) {
  if(s.hasOwnProperty(p)) {
    if(a.hasOwnProperty(p) && !b.hasOwnProperty(p))
      r[p] = a[p];
    else if(b.hasOwnProperty(p) && !a.hasOwnProperty(p))
      r[p] = b[p];
    else
      r[p] = c;
  }
}
```

Property	TAJS	HOO
$p \in Fr \rightarrow p \in Fa \cup Fb \cup F_1$	✗	✓
$p \in Fa \rightarrow p \in Fr$	✗	✓
$p \in Fb \rightarrow p \in Fr$	✗	✓
$p \in Fa \wedge p \in Fb \rightarrow \texttt{r}[p] = v_4$	✗	✓
$p \in Fa \wedge p \notin Fb \rightarrow \texttt{r}[p] \in V_2$	✗	✓
$p \notin Fa \wedge p \in Fb \rightarrow \texttt{r}[p] \in V_3$	✗	✓
$p \in Fr \wedge p \notin Fa \wedge p \notin Fb \rightarrow \texttt{r}[p] \in V_1$	✗	✓

Listing D.5 – Merge

$[\mathtt{s} \mapsto \{a_2\} * \{a_2\} \cdot \langle F_2 \mapsto V_2 \rangle * \mathtt{t} \mapsto \{a_3\} * \{a_3\} \cdot \langle F_3 \mapsto V_3 \rangle]$

```
var r = {};
for(var p in a) {
  if(a.hasOwnProperty(p))
    r[p] = a[p];
}
for(p in b) {
  if(b.hasOwnProperty(p))
    r[p] = b[p];
}
```

Property	TAJS	HOO
$r \neq a$	✓	✓
$r \neq b$	✓	✓
$p \in Fr \rightarrow p \in Fa \cup Fb$	✗	✓
$p \in Fa \rightarrow p \in Fr$	✗	✓
$p \in Fb \rightarrow p \in Fr$	✗	✓
$p \in Fb \rightarrow \mathtt{r}[p] \in V_3$	✗	✓
$p \in Fa \wedge p \notin Fb \rightarrow \mathtt{r}[p] \in V_3$	✗	✓

E. Detailed Tests for Two-State HOO with Attribute/Value Trackers and Desynchronization

This chapter contains the open-object-focused JavaScript programs that were analyzed as part of the evaluation in Chapter 6. Specifically, it focuses on four categories of benchmarks: classes, memoization, mixins, and traits. Each of the analyzed examples are given in two parts. The first part is the code that was analyzing with possibly a small amount of context. The second part is a precondition for the analysis.

The preconditions that are given are specified in a special language that documents the structure of the heap at the beginning of the function. There are four main parameters in each precondition: `global`, ***this***, `closure`, and `arguments`, which represent the global object, the object pointed to by the ***this*** keyword, the closure object, and the object containing arguments respectively. Inside these objects, there specifications for objects. Single quotes specify constant attribute names. Attribute names without single quotes specify sets of unknown cardinality. The`--` specifier gives a handle to an object that cannot be accessed. This is used for specifying parts of the heap that the function does not make use of.

Note that the preconditions are written in a form that mimics the internal structure of the program after preprocessing. Consequently, variables that are captured by the closure require an extra pointer dereference to access. This means that there can be many empty string attribute names, which represent simple pointers.

There is a single classes example that is synthetic. The reason for using a synthetic benchmark here as opposed to one or more of the myriad of class systems available for JavaScript is the number of language features used by real class systems exceeds the capabilities of the JSAna analyzer today. Instead a synthetic class example is used that incorporates many of the features of the real class systems used by JavaScript. Note that it does not rely upon prototype-based inheritance to implement the class system like many libraries do. The primary reason for this is that adding prototype support for this purpose is simple and not overly problematic. Object copies (which are still often used in the prototype-based class systems) are quite problematic, however, so the example folds in that functionality. This example is shown in Listing E.1.

There are both synthetic and non-synthetic memoization examples. The synthetic example (Listing E.2) is similar in structure to the real code available in the Google Closure library (version 5fbc334ce7f86018da9fb5e673d2518143aa999e) shown in Listing E.4. The primary difference is the number of options that are made available in the code. However, to analyze the code offered in the Google Closure library, some small changes had to be

Listing E.1 – Class (synthetic)

```
var Class = function(cfg) {
  // outer copy
  var copy = function(res,src,exc) {
    for(var p in src) {
      if(p in exc) {
      } else {
        res[p] = src[p];
      }
    }
  };
  // save the configuration
  var attrs = {};
  copy(attrs,cfg,{});
  // save off init
  var init = cfg.init;
  // construct the result
  var result = function()
    //<precondition>
    @pre [
      global = {};
      this = global;
      closure = {
        'init': {'': [@init,{}]},
        'attrs': {'': {Fattrs: --}}
      };
      arguments = {Fo: --};
    ]
    //</precondition>
  {
    // inner copy
    var copy = function(res,src,exc) {
      for(var p in src) {
        if(p in exc) {
        } else {
          res[p] = src[p];
        }
      }
    };
    var result = {}
    copy(result,attrs,{init:""});
    init(result, arguments);
    var rv = result;
  };
}
```

made from the original shown in Listing E.3. The biggest changes are due to missing features in the analysis. The standard technique of using falsiness and the logical-or operator is unsupported right now. As a result, one option is hard coded. Additionally, the ternary operator is unsupported, so it is replaced with a normal if-then-else structure. Finally, the return statement is omitted and the result is accessed through the `result` variable.

The mixins example is exactly the `extend` function from the Prototype.js library version 1.7.2 (Listing E.5). This function uses very few features of the language and does not need additional simplifications.

The final benchmark is an implementation of the composition function for an implementation of traits. There are two versions of this considered: a synthetic version and an adapted version of the `compose` function from Traits.js. The functions are quite similar in structure, but once again, there are many more options available in the Traits.js code. Listing E.6 shows the synthetic version of traits made by extending the mixins functionality to the purpose of constructing traits. Listing E.7 shows the version of traits composition that is provided by Traits.js from the date October 24, 2011. However, this version differs in a number of ways from the adapted version in Listing E.8. The adaptations once again come from the lack of support for certain features. For example, array support is limited, so this benchmark unrolls the outer loop to check a binary compose. Functions are inlined to work around a bug that sometimes affects calls to known functions. Additionally, some branches are eliminated that depend on functionality (such as booleans) that are not directly supported by the analysis at this time. Further, the conflict value generator is replaced with a conflict value 'conflict' because the conflict value generator relies upon unsupported object and string operations.

Listing E.2 – Memo (synthetic)

```
var memo = function (f)
{
  // memoization table
  var memo = {};
  var memoized = function()
    //<precondition>
    @pre [
      global = {
        'get_key': [@gk,{}]
      };
      this = global;
      closure = {
        'memo': {'': {Fm: --}},
        'f': {'': [@f,{Ff: --}]}
      };
      arguments = {...};
    ]
    //</precondition>
  {
    var result;
    // get the unique identifier
    var key = get_key(arguments);

    if(key in memo) {
      // result is in memoization table
      result = memo[key];
    } else {
      // result is not in memoization table
      result = f(arguments);
      memo[key] = result;
    }
  }
}
```

Listing E.3 – Google Closure memoize (original with modified line wraps)

```
goog.memoize = function(f, opt_serializer) {
  var serializer = opt_serializer ||
                   goog.memoize.simpleSerializer;

  return function() {
    if (goog.memoize.ENABLE_MEMOIZE) {
      // In the strict mode, when this function is called as a
      // global function, the value of 'this' is undefined
      // instead of a global object. See:
      // https://developer.mozilla.org/en/JavaScript/Strict_mode
      var thisOrGlobal = this || goog.global;
      // Maps the serialized list of args to the corresponding
      // return value.
      var cache = thisOrGlobal[goog.memoize.CACHE_PROPERTY_] ||
          (thisOrGlobal[goog.memoize.CACHE_PROPERTY_] = {});
      var key = serializer(goog.getUid(f), arguments);
      return cache.hasOwnProperty(key) ? cache[key] :
          (cache[key] = f.apply(this, arguments));
    } else {
      return f.apply(this, arguments);
    }
  };
};
```

Listing E.4 – Google Closure memoize (modified)

```
(function () {
  var serializer;
  var f;

  return function()
  //<precondition>
  @pre [
    global = {
      'goog': {
        'memoize': {
          'ENABLE_MEMOIZE': {Fe: --},
          'CACHE_PROPERTY_': 'cache'
        },
        'getUid': [@uid,{}]
      }
    };
    this = { };
    closure = {
      'f': {'': [@f,{}]},
      'serializer': {'': [@serializer,{}]}
    };
    arguments = {Fo: --};
  ]
  //</precondition>
  {
    var result;
    if ('enable' in goog.memoize.ENABLE_MEMOIZE) {
      var thisOrGlobal = this;
      var cache;
      if(goog.memoize.CACHE_PROPERTY_ in thisOrGlobal) {
        cache = thisOrGlobal[goog.memoize.CACHE_PROPERTY_];
      } else {
        cache = {};
        thisOrGlobal[goog.memoize.CACHE_PROPERTY_] = cache;
      }
      var key = serializer(goog.getUid(f), arguments);
      if(key in cache) {
        result = cache[key];
      } else {
        result = f(this, arguments);
        cache[key] = result;
      }
    } else {
      result = f(this, arguments);
    }
  };
});
```

Listing E.5 – Prototype.js extend (mixins)

```
var extend = function (destination,source)
  //<precondition>
  @pre [
    global = {};
    this = global;
    closure = {};
    arguments = [
     /* destination */ {Fo: --},
     /* source      */ {Fs: --}
    ];
  ]
  //</precondition>
{
  // iterate over attributes doing copy
  for(var property in source) {
    destination[property] = source[property];
  }
}
```

Listing E.6 – Traits compose (synthetic)

```
var compose = function (a,b)
  //<precondition>
  @pre [
    global = {};
    this = global;
    closure = {};
    arguments = [
     /* a */ {Fo1: --},
     /* b */ {Fo2: --}
    ];
  ]
  //</precondition>
{
  // create result object
  var r = {};
  // create conflict value
  var c = "conflict";
  // iterate through a placing
  // conflicts and a values into r
  for(p in a) {
    if(p in b) {
      r[p] = c;
    } else {
      r[p] = a[p];
    }
  }
  // iterate through b placing
  // values into r
  for(p in b) {
    if(p in a) {}
    else {
      r[p] = b[p];
    }
  }
}
```

Listing E.7 – Traits.js compose (original with modified line wraps)

```
function compose(var_args) {
  var traits = slice(arguments, 0);
  var newTrait = {};

  forEach(traits, function (trait) {
    forEach(getOwnPropertyNames(trait), function (name) {
      var pd = trait[name];
      if (hasOwnProperty(newTrait, name) &&
        !newTrait[name].required) {

          // a non-required property with the same name was
          // previously defined this is not a conflict if pd
          // represents a 'required' property itself:
          if (pd.required) {
            return; // skip this property, the required
            // property is now present
          }

          if (!isSameDesc(newTrait[name], pd)) {
            // a distinct, non-required property with the same
            // name was previously defined by another
            // trait => mark as conflicting property
            newTrait[name] = makeConflictingPropDesc(name);
          } // else,
          // properties are not in conflict if they refer to
          // the same value

        } else {
          newTrait[name] = pd;
        }
    });
  });

  return freeze(newTrait);
}
```

Listing E.8 – Traits.js compose (modified)

```
var compose = function ()
  //<precondition>
  @pre [
    global = {};
    this = global;
    closure = {};
    arguments = [
      {Fo1: --},
      {Fo2: --}
    ];
  ]
  //</precondition>
{
  var traits = arguments;
  var newTrait = {};
  var trait;

  trait = traits['0'];
  for(var name in trait) {
    var pd = trait[name];
    if (name in newTrait) {
      newTrait[name] = "conflict";
    } else {
      newTrait[name] = pd;
    }
  }
  trait = traits['1'];
  for(var name in trait) {
    var pd = trait[name];
    if (name in newTrait) {
      newTrait[name] = "conflict";
    } else {
      newTrait[name] = pd;
    }
  }
}
```

F. Detailed Tests for QUIC Graphs

This chapter contains the raw input files for use with the QUIC Graphs demonstration program. These are the result of hand translating the corresponding tests from the Python test suite from Python version 2.7.3. The language used here is intended to look like JavaScript. The key difference is that curly braces denote sets instead of objects and the language is strongly typed.

The operators include union `|` and common math operators for addition subtraction and comparison. The assertion language overloads the comparison operators for both numbers and set operations.

Listing F.1 – copy

```
var s: set;
var x: num;
var r: set;

r = {};

for(x in s) {
  r = r | {x};
}

assert(r <= s);
assert(s <= r);
```

Listing F.2 – filter

```
var s: set;
var r: set;
var x: num;
var t: num;

r = {};

for(x in s) {
  if(x > 10) {
    r = r | {x};
  }
}
assert(r <= s);
t = choose(r);
assert(t > 10);
```

Listing F.3 – generic_max

```
var max: num;
var x: num;
var d: set;

max = 0;

for(x in d) {
  assume(x < 100);
  if(x > max) {
    max = x;
  }
}

d = d - {max};

for(x in d) {
  assert(x <= max);
  assert(x < max);
}
```

Listing F.4 – merge

```
var x: num;
var y: num;
var l: set;
var u: set;
var r: set;
var ref: set;

r = {x};
for(y in l) {
  r = r | {y};
}

for(y in u) {
  r = r | {y};
}

ref = {x} | l | u;

assert(r <= ref);
assert(ref <= r);
```

Listing F.5 – partition

```
var s: set;
var l: set;
var u: set;
var x: num;
var y: num;

l = {};
u = {};
x = choose(s);
for(y in s) {
  if(y < x) {
    l = l | {y};
  } else {
    if (y > x) {
      u = u | {y};
    }
  }
}
assert(choose(l) < x);
assert(choose(u) > x);
assert(l <= s);
assert(u <= s);
```

Listing F.6 – test_builtin_filter

```
var seq: set;
var res: set;
var x: num;


res = {};
for(x in seq) {
  if(x == 0) {
    res = res | {x};
  }
}
assert(choose(res) >= 0);
assert(choose(res) <= 0);
```

Listing F.7 – test_builtin_map

```
var d: set;
var r: set;
var x: num;
var i: num;

d = {};
i = 1;
while(i <= 3) {
  d = d | {i};
}

r = {};
for(x in d) {
  r = r | {x+7};
}

assert(choose(r) <= 10);
assert(choose(r) >= 8);
```

Listing F.8 – test_builtin_max_min

```
var min: num;
var max: num;
var seq: set;
var x: num;

min = 0;
max = 1;
for(x in seq) {
  if(x < min) {
    min = x;
  }
  if(x > max) {
    max = x;
  }
}
assert(max > min);
```

Listing F.9 – test_builtin_reduce

```
var d: set;
var s: num;
var x: num;

assume(choose(d) >= 1);
assume(choose(d) <= 3);

s = 0;
for(x in d) {
  s = s + x;
}

assert(s >= 1);
```

Listing F.10 – test_difference

```
var s1: set;
var s2: set;
var s3: set;
s3 = s1 - s2;
assert({choose(s3)} <= s1);
```

Listing F.11 – test_has_key

```
var d: set;
var x: num;
var y: num;

d = {};
if({1} <= d) {
  assert(1 <= 0);
} else {
}


d = {x, y};
assert(d == {x, y});
```

Listing F.12 – test_intersection

```
var s1: set;
var s2: set;
var s3: set;
s3 = s1 & s2;
assert({choose(s3)} <= s1);
assert({choose(s3)} <= s2);
```

Listing F.13 – test_iter_independence

```
var seq: set;
var res: set;
var i: num;
var j: num;
var k: num;
var sum: num;

res = {};
for(i in seq) {
  assume(i >= 0);
  for(j in seq) {
    assume(j >= 0);
    for(k in seq) {
      assume(k >= 0);
      sum = i+i+i+i+i+i+i+i+i + j+j+j + k;
      res = res | {sum};
    }
  }
}
assert(choose(res) >= 0);
```

Listing F.14 – test_keys

```
var d: set;
var x: num;
var y: num;
d = {};
assert(d == {});
d = {x, x};
assert({x} <= d);
assert({y} <= d);
```

Listing F.15 – test_multi_return

```
var d: set;
var i: num;
var n: num;
assume(n > 0);
d = {};
i = 0;
while(i < n) {
  d = d | {0};
  i = i + 1;
}
assert(choose(d) <= 0);
assert(choose(d) >= 0);
```

Listing F.16 – test_nested_dependent

```
var d: set;
var s: num;
var x: num;
var y: num;

s = 0;
for(x in d) {
  for(y in d) {
    if(y < x) {
      s = s + x + y;
    }
  }
}
assert(s >= 0);
```

Listing F.17 – test_pop

```
var d: set;
var s: set;
var t: set;
var x: num;
var y: num;
d = {};
d = d | s;
x = choose(d);
d = d - s;
assert({x} <= s);
d = d | t;
assume(x == x);
y = choose(d);
assume(x == x);
assert({y} <= t);
```

Listing F.18 – test_resize1

```
var d: set;
var i: num;
var n: num;
var m: num;

assume(n > 0);
assume(m > n);

d = {};
i = 0;
while(i < n) {
  d = d | {i};
  i = i + 1;
}

assert(choose(d) >= 0);
assert(choose(d) <= n);

i = 0;
while(i < n) {
  d = d - {i};
  i = i + 1;
}

assert(d == {});

i = n;
while(i < m) {
  d = d | {i};
  i = i + 1;
}

assert(choose(d) >= n);
assert(choose(d) <= (m + -1));
```

Listing F.19 – test_simple_conditional

```
var d: set;
var i: num;
var s: num;
var x: num;
var n: num;

d = {};
i = 0;
while(i < n) {
  d = d | {i};
  i = i + 1;
}
assert(choose(d) >= 0);
assert(choose(d) < n);

s = 0;
for(x in d) {
  if(x < 50) {
    s = s + x;
  }
}
assert(s >= 0);
assert(s <= 1225);
```

Listing F.20 – test_simple_nested

```
var r: set;
var x: num;
var y: num;
var s1: set;
var s2: set;
var a: num;
var b: num;
r = {};
s1 = {a};
s2 = {b};
for(x in s1) {
  for(y in s2) {
    r = r | {x + y};
  }
}
assert(choose(r) <= (a+b));
assert(choose(r) >= (a+b));
```

Listing F.21 – test_simpler_nested

```
var r: set;
var x: num;
var y: num;
var s1: set;
var s2: set;
var a: num;
var b: num;
r = {};
s1 = {a};
s2 = {b};
for(x in s1) {
  for(y in s2) {
    r = r | {x + y};
  }
}
assert(choose(r) <= (a+b));
assert(choose(r) >= (a+b));
```

Listing F.22 – test_srange

```
var d: set;
var n: num;
var i: num;

d = {};
i = 0;
while(i < n) {
  d = d | {i};
  i = i + 1;
}
assert(choose(d) < n);
assert(choose(d) >= 0);
```

Listing F.23 – test_union

```
var s1: set;
var s2: set;
var s3: set;
s3 = s1 | s2;
assert({choose(s1)} <= s3);
assert({choose(s2)} <= s3);
```