TU WIEN Informatics

Join-Operatoren für die bi-abduktive Analyse von Low-Level-Code

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Lukas Rysavy, BSc
Matrikelnummer 11809637

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Mitwirkung: Univ.Ass. Florian Sextl, MSc

Wien, 2024-04-03

Lukas Rysavy | Florian Zuleger

TU WIEN Informatics

Join Operators for Bi-Abductive Analysis of Low-Level Code

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Lukas Rysavy, BSc
Registration Number 11809637

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Math. Dr.techn. Florian Zuleger
Assistance: Univ.Ass. Florian Sextl, MSc

Vienna, 2024-04-03

Lukas Rysavy Florian Zuleger

Erklärung zur Verfassung der Arbeit

Lukas Rysavy, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2024-04-03

Lukas Rysavy

Danksagung

Zuerst möchte ich mich bei meinem Betreuer, Prof. Florian Zuleger, für die mir von ihm zur Verfügung gestellten Freiheiten, Dinge selbst zu probieren, aber auch für den wertvollen Input bei Unklarheiten, bedanken. Weiters möchte ich mich bei meinem Assistenzbetreuer Florian Sextl für immer zeitnahe Antworten auf zahlreiche Fragen und damit für die Klärung von vielen Problemen bedanken.

Außerdem möchte ich Dank an das Broom-Team der Technischen Universität in Brno ausrichten, welches sehr hilfreich beim Verstehen des Codes von Broom war; speziell sind hier Adam Rogalewicz und Veronika Šoková hervorzuheben.

Zuletzt möchte ich Danke an meinen besten Freund und Kollegen Manuel Reinsperger sagen, welcher mir meine ganze Zeit an der Universität beiseite gestanden ist und beim Korrekturlesen dieser Arbeit geholfen hat.

Acknowledgements

First, I want to thank my supervisor Prof. Florian Zuleger for giving me the freedom to try things by myself while giving valuable input and guiding me in unclear situations. Furthermore, I want to thank my assistant supervisor Florian Sextl for always responding to any questions I had in minimal time and like that resolving many uncertainties I encountered.

In addition, I want to thank the Broom team from the Brno University of Technology, especially Adam Rogalewicz and Veronika Šoková, for providing remote help on the inner workings of Broom.

Lastly, I want to thank my best friend and colleague Manuel Reinsperger for being there with me throughout all my years in university and proofreading this thesis.

Kurzfassung

Statische Codeanalyse spielt eine wichtige Rolle bei der Sicherstellung von Korrektheit bestimmter Programme oder von Segmenten davon. Bewerkstelligt wird dies oft mittels abstrakter Interpretation, bei der Verträge, bestehend aus Vor- und Nachbedingungen, für Codeblöcke generiert werden. Eine in diesem Kontext offene Frage ist, wie Situationen, in denen zwei Programmpfade aufeinandertreffen (häufig am Ende von Fallunterscheidungen oder Schleifen), und in denen mehrere solche Verträge verfügbar sind, gehandhabt werden sollen. So ist diese Frage auch in `Broom`, einem Werkzeug für die statische Analyse speziell von systemnahem Code, welcher mit verketteten Datenstrukturen arbeitet, bislang unbeantwortet.

Aus diesem Grund wird ein Algorithmus, welcher es ermöglicht, von Broom erzeugte Formeln zu vereinen, vorgestellt, und seine Auswirkungen auf die Qualität der Ergebnisse sowie die Effizienz des Analyseprozesses werden durch einen Vergleich von Verträgen, welche für repräsentative Beispiele mit und ohne dieses Join-Algorithmus generiert werden, untersucht. Mit den Ergebnissen dieser Untersuchungen werden schlussendlich Parameter für den Algorithmus gefunden, welche zu möglichst guten Resultaten führen.

Abstract

Static code analysis plays an important role in ensuring the correctness of program segments or entire programs, often performed using abstract interpretation to generate contracts consisting of pre- and postconditions for code blocks. Here, an ongoing question is how to handle situations where two program paths merge (commonly at the end of conditional branches or loops) and several such contracts are available. Specifically, this question has previously gone unanswered in `Broom`, a static analysis tool that specializes in analyzing low-level code that operates on linked pointer segments.

For that reason, an algorithm capable of joining the formulas generated by Broom is introduced, and its effects on the quality of generated results as well as on the efficiency of the analysis process are studied by comparing contracts generated for a set of representative examples while using this join algorithm to ones generated without. With the conclusions of this evaluation, the parameters for the proposed algorithm are adjusted to get the best possible results.

Contents

CHAPTER 1

Introduction

To automatically prove certain properties of programs such as them being error-free or whether some assertions hold after their execution, static analysis is employed in many cases, often executing the program symbolically to find generalized pre- and postconditions for code blocks without the need to perform all of the calculations on concrete values.

This approach is, among many others, taken by the project Broom[HPR$^+$22], which this thesis is based upon. Broom is a static analysis tool for C programs that especially aims to support low-level memory operations such as pointer arithmetics or memory reinterpretation.

1.1 Problem Setting

When symbolically executing a program, Broom adds the pre- and postcondition of the currently executed statement to the list of missing preconditions and the current state's postcondition, respectively, by solving the bi-abduction problem. The bi-abduction question addresses the problem of finding the antiframe and the frame of an entailment: The antiframe consists of a formula that needs to be added to the left side of the entailment to make it valid, while the frame is any state that is unchanged over the course of the entailment and can be added on its right side. Naturally, the antiframe to be found should be as small as possible while the frame should contain as much information as possible.

Using the answers to this question, the precondition and postcondition of a function can be incrementally built up statement by statement. Broom traverses functions along the call tree, starting from the very bottom and working its way upwards. This way, contracts for function call statements are available at the time they are analyzed, and they can be treated just like any other instruction. For this reason, Broom does not support recursive function calls. On the other hand, this also enables the possibility of analyzing open programs, ie. programs where not all parts are available at the time of analysis (e.g. shared libraries that cannot be executed on their own but rather serve as a pluggable part for other programs).

To represent operations on the heap of a program, standard Hoare logic is extended with two operators from separation logic[Rey02]. The **star operator `*`** ensures that two clauses lie in different parts of the heap and thus deals with the aliasing question. The **points-to operator** $\mapsto$, on the other hand, is a predicate indicating a pointer to a specific address in memory.

When analyzing non-linear programs containing e.g. loops, abstraction is of utmost importance to speed up or even enable the termination of the analysis altogether. Since linked pointer segments play a central role in Broom, it seems natural to make them the unit of abstraction. Therefore, Broom adds predicates for singly- and doubly-linked pointer segments, containing information about the segment's start and end as well as generalized information about the individual links, including how to get from one link to the next. These predicates are the result of an abstraction procedure applied at join points (e.g. the end of a list) that heuristically tries to find pointer chains with compatible elements.

1.2 Problem Statement

By default, loop termination in Broom is achieved by opportunistically abstracting linked segments and checking for entailment between states, pruning any that are overruled by one or more others. This is based on the assumption that after a certain number of iterations, no new information will be added to the pre- and postcondition formulas that cannot be collapsed into any of the existing segments, in which case a fixed point has been reached.

For example, take the function in Listing 1.1, with the discovered postconditions in the comments: In the beginning, only the fact that the `list` variable has not changed is known (ie. equivalent to its anchor, the value at the start of the function). Then, its inequivalence to `NULL` in pre- and postcondition is discovered, and for each iteration, an additional pointer predicate is added (lines 11 and 12). This is abstracted into a singly-linked segment predicate (`Slseg`, line 14) which entails both of the previous states and is stable through the next iteration. Preconditions are found similarly, with the `list` variable replaced by a logical one.

For many cases, this works just as expected and reaches a stable state after just a few loop iterations (the default limit in Broom is five iterations before the analysis surrenders). However, there are also quite a few examples where this needs more iterations than technically necessary with a "perfect" algorithm and where the analysis therefore takes more time, or where no result can be found at all after a sensible number of loop iterations. Certainly, there is room for improvement in this part of the analysis process.

Listing 1.1: Example linked list loop

```
struct linked_list {
  struct linked_list* next;
  ...
}

struct linked_list* linked_list_last(struct linked_list* list) {
  // { list = list_anch }
  while (list != NULL) {
    // { list = list_anch & list != NULL }
    list = list->next;
    // { list_anch-(8)->list & list_anch != NULL }
    // { list_anch-(8)->%l1 * %l1-(8)->list & list_anch != NULL }
  }
  // { Slseg(list_anch, list) & list_anch != NULL & list = NULL }
  return list;
}
```

Therefore, there is a demand for a join operator that takes several states (each with a pre- and a postcondition) and either joins them into a single, more generic version or suggests that such a join is not possible without a significant loss of information. In addition to finding such an operator that can improve analysis results in such suboptimal cases and find contracts where this might have been not possible previously, an evaluation of the said operator is desired, showing cases where such an improvement can be achieved and cases where this is not possible, possibly together with explanations for both.

1.3 State-of-the-Art

The problem at hand has been solved for numerous analyzers in various ways, each solution with its advantages and drawbacks. Many introduce some kind of normalized form for their formulas that can more easily be compared and parts matched up with each other; for example, the approach presented by Li et al.[LBCR17] transforms concrete memory graphs into so-called silhouettes represented by regular expressions, which can then simply be generalized to serve as a summary of several of the states.

A project that deserves special acknowledgment is Predator[DPV11, DPV13, DPV14, HKP$^+$16], one of the main inspirations for Broom. In Predator, states are represented using Abstract Memory Graphs, which are built via symbolic execution. Predator includes a very powerful join operator that can join such graphs, with a scope even larger than that of the operator introduced in this thesis: Not only does the join prune superfluous states, but is also the only component in the system that performs abstraction on the states and is even responsible for finding entailment relations. In addition to the differing state structure, one major difference between Predator and Broom is that Predator can only handle closed programs, ie. programs where code is available for all relevant functions.

1.4 Methodology

To solve this problem in the context of the Broom analyzer, a join operator has been implemented in OCaml and incorporated into the system. The implementation and its design decisions were guided by the result of analyses of real-world code examples: The performance of the analysis was evaluated for various use cases, and rules on how to join which states were derived. In addition, the behavior of the join process of other projects such as Predator has been analyzed and used as an inspiration for many parts of the implementation presented here.

The finished implementation was experimentally evaluated on a rather comprehensive test set of relevant code pieces. Here, multiple factors such as the analysis runtime and the quality of its results were compared. For mismatches (e.g. cases where more contracts could be generated or the runtime differed significantly), the individual join operations (with their input and output states) were analyzed by hand, finding any changes that were introduced to the resulting state and how this affected future iterations of the symbolic execution.

1.5 Contributions

1.5.1 The Join Operator

This thesis presents a join operator that first converts formulas to so-called “types”, with information grouped by the involved variables (despite their name not connected to C types, but only containing information gained from the semantics of the code’s operations). Types include, for example, pointers or integer scalars, each possibly with additional information about the variable (like a concrete integer value, or the variables they point to at certain offsets). In addition to types, each variable can have a set of constraints, representing information not specific to a certain type such as the size of the containing memory block or equivalences to other variables.

These types are extracted from pre- and postconditions using a simple set of rules and can be converted back just as easily. However, they form a much better basis for structured joining since this is done on a per-variable basis. With a variable-type map, all information for a specific variable is easily available in a structured way without the need to look through the whole formula.

Types are then joined recursively, starting with program variables. Then, depending on the kinds of types of the two variables to join, different rules are applied, each either returning a new, possibly more general type, or a failure token to make the respective path of the join fail. When joining pointer types, their targets are joined pairwise, updating the respective types on the fly.

Listing 1.2: Types for the example states (shortened)

```
list_anch: {
  // state 1:
  0-(8)->list,
  // state 2:
  0-(8)->%l1,
}
%l1: {
  0-(8)->list
}
list: ?
```

In addition, pointers can be collapsed into pointer chain segments, and the algorithm even includes an abstraction procedure where pointer chains are abstracted into a segment predicate if a join is otherwise not possible (for example, because one segment ends sooner than the other one). For these segments, the join algorithm is called recursively as well, joining variables inside their lambdas (the description of a single link in the segment, including the pointer to the next link).

Similarly, constraints are joined for each encountered variable via a simple set of rules. Here, the algorithm checks whether any additional equivalences between previously otherwise possibly distinct variables need to be introduced for the join to succeed and fails if that is the case.

After a successful join, variable types are converted back to formulas, and a check for entailment is performed using a solver to filter out any potentially unsound join results (generated because of choices made for the sake of simplicity and performance). If the entailment holds from the new to the old states as well, a fixed point has been found and the corresponding loop can terminate.

This procedure is performed on all pairs of old and new states (from before and from the respective loop iteration), taking states that were successfully joined out of the equation. There might be some improvements to this, checking for compatibility before the actual join, but this is not the main topic of this thesis and is left open for future research.

In the example from Listing 1.1, where the types for the two states in lines 11 and 12 are shown in Listing 1.2 (shortened to the relevant parts), the two states would be joined as follows: `list_anch` has a pointer type that points to `%l1` and `list`, so these two variables are joined. Since the former is a pointer and the latter is unknown (`list` is the last known link, so the analysis does not know whether there are more pointers from there), the join cannot immediately succeed and abstraction is performed, yielding the same linked segment as before. All pointers are collapsed into this segment and the state is stable, leading to the correct result being returned.

1.5.2 Evaluation

The performance of this algorithm has been evaluated and compared to the baseline implementation in Broom where abstraction happens opportunistically and states are only pruned where an entailment holds. For this comparison, multiple aspects have been considered, including the runtime and memory usage of the analysis, the number of join points encountered, and the quantity and quality of generated contracts.

Runtime and memory usage have been found to generally be higher when using the join operator, consistent with the fact that the number of required join points only differs very marginally. While the number of contracts that can be generated for each function is very similar to the base case as well, there are several functions where contracts are either only found when the join operator is used or the limit of loop iterations is increased, indicating a general improvement of analysis results caused by the operator. In addition, for some functions, the join operator contributes to semantically equivalent but more concise contracts, presumably due to the normalization applied during the join.

This thesis first discusses related work in Chapter 2 and then dives deeper into the ecosystem around Broom in Chapter 3. Then, requirements for a join algorithm are established in Chapter 4 and the algorithm together with its implementation for Broom are described in Chapter 5 and 6, respectively, together with possible variations in Chapter 7. Finally, experimental results comparing the algorithm to the base case are outlined and discussed in section 8.

CHAPTER 2

Related work

Broom, which the work in this thesis is based upon, and its surrounding ecosystem have been described in [HPR$^+$22, Kai23, SRVZ23]. Broom is a static analysis tool for C code that specializes in analyzing code that utilizes low-level memory operations such as pointer arithmetic and in particular supports linked pointer segments and data structures that make use of them, like linked lists. This is accomplished by always operating on a per-field basis and providing abstraction predicates that represent singly- or doubly-linked pointer chain segments of a certain structure.

Broom uses bi-abductive analysis to generate contracts for functions that consist of pre- and postconditions expressed using separation logic. Code is executed symbolically, finding missing preconditions and the current state's postconditions by solving the bi-abduction problem. This is done for all functions along the call tree, starting from the leaves, to retrieve contracts for all available functions. It is worth noting that Broom also supports open programs, ie. programs where not all parts are available at the time of analysis; there, contracts can still be generated for any available functions. This functionality has been described extensively in the aforementioned papers and is detailed in Section 3. Furthermore, Broom's source code is available at [HPR$^+$19].

A similar system that served as a big inspiration for Broom, uses a comparable abstraction model and includes a join operator, although for slightly differently structured states, is Predator[DPV11, DPV13, DPV14, HKP$^+$16]. Predator uses a model called Symbolic Memory Graph, which is a graph that represents an abstract state of the program's heap. Such graphs consist of objects and values, representing allocated objects or memory regions as well as concrete values, respectively. In addition, there are edges between these entities, describing either has-value or points-to relations. Objects can not only be predefined single-size memory regions but also list segments or 0/1 abstract objects that are possibly NULL.

The join operator in Predator is a core component of the system, with it not only joining two symbolic memory graphs to prune the state space but also serving as the location for abstraction as well as even a basis for entailment. Joining at loop edges is therefore strictly required for the termination in Predator, with the final state emanating from the join result of multiple loop iterations. Predator's functionality is outlined in more detail in section 4. Sadly, formal documentation of the inner workings of the said operator is limited, and so Predator's code[DVP$^+$16] itself must provide additional details when analyzing the rules used to join memory graphs.

Shape analysis, with its goal of analyzing properties of programs that work with heaps and pointers therein, has been a popular "genre" of program analysis for many years; the logic used to accomplish this in Broom, separation logic, has been formally described by [Rey02]. Since then, a lot of work has been based on separation logic, including, among others, [BCC$^+$07, BBC08, Atk10, BIP10, Chl11, DOY06].

Some have, just like Broom, extended this logic with predicates indicating pointer segments or linked-list-like structures to act as a unit of abstraction and thus as the basis for the termination of the analysis; examples are [CRN07, DOY06].

Primarily, separation logic, as an instance of the logic of bunched implications[OP99], introduces two additional operators besides the ones available in standard Hoare logic: The separating conjunction and the points-to operator. While the former provides a way to split a formula into several parts that each operate on a distinct, non-overlapping subset of the heap, the latter defines the formula counterpart to memory addresses stored as pointers in C. Separation logic is explained in Section 3.2.

Just like in Broom, bi-abductive inference is often used to synthesize separation-logic-based contracts, including pre- and postconditions, from code blocks[CDOY11, TLDC13, LGQC14, CLQ19, CL20]; the term was popularized by Calcagno et al.[CDOY11]. Bi-abduction is usually used as part of the symbolic execution pipeline and enables the analyzer to simultaneously find missing parts of the precondition needed to execute a piece of code and the state that holds after its execution by solving the abduction problem as well as the frame problem, respectively. This way, program statements can be analyzed one by one to find contracts for code blocks and whole functions. Section 3.1 introduces bi-abduction formally and explains how it can be used in the analysis process.

Likewise, low-level pointer operations have been subject to research[KSV10, Min06, CDOY06], analyzing the goal of performing shape analysis of programs with unstructured memory access, including things such as pointer arithmetic or block splitting. The problems and complications these operations can pose for such an analysis are detailed in section 3.5; many different solutions have been proposed, most dictating some of the main design choices when building such an analyzer.

Apart from the documentation on joining in Predator, there has been some work on joining states described using separation logic, although the literature on this topic is relatively scarce. Work from these papers is discussed further in section 4.

[LBCR17] introduces an approach that calculates the silhouette of a symbolic heap and checks for compatibility of several such silhouettes to find join candidates to later generalize their silhouettes into a common match. A silhouette is a set of vertices and paths, corresponding to variables and pointers, respectively, represented as a regular expression (enabling repetition to symbolize repeated segments). These regular expressions can be compared easily to find matching states, which are then joined by generalizing the expressions into a common, more generic version. This approach does not only suggest an algorithm for the join itself, but also for the process of selecting sets of states that match and can therefore be joined with each other.

[YLB$^+$08], on the other hand, proposes a rule-based join operator that reduces input states and simultaneously adds generalized (abstracted) versions of what was removed to the output state. Later, the amount of information lost because of the join is determined to check whether the result is accepted.

To accomplish this, a set of rules is applied incrementally to build the resulting state and auxiliary information while simultaneously chipping away parts of the input states until they are empty and the result contains all necessary information. Auxiliary information includes an equivalence map of joined variables as well as lists of segments that have been generalized and are now possibly empty. Using this information, a check is performed to possibly find equalities between variables introduced in the course of the join that are not present in both input states, which indicates a high loss of information and therefore causes the join to fail.

Lastly, [MSRF04] defines an abstraction operator to merge states that have the same universe while possibly keeping several disjunct states, very similar to what is done in this thesis. Determining universe congruence equivalence classes is accomplished by converting concrete program configurations to abstract ones, expressed in 3-valued logic. This way, discrepancies between two states are expressed with the 1/2 value made available by 3-valued logic, which enables non-deterministic choices for values and pointers. Using this abstraction layer, matching states can easily be joined into a common higher-level state. An abstraction function is defined that performs this operation in a bounded manner so as not to abstract too much and get a trivial state.

CHAPTER 3

Broom

The open-source project Broom[HPR$^+$22], a “static analyzer for C based on separation logic and the principle of bi-abductive reasoning”[HPR$^+$19], was developed as a collaboration between the Brno University of Technology and the Vienna University of Technology and aims to statically analyze C programs that utilize low-level pointer operations, with a special focus on linked pointer segments. This chapter dives into the functionality of Broom, as well as the paradigms used in the project.

Contracts are generated by Broom using bi-abduction (see section 3.1) and expressed in separation logic (see section 3.2). Since linked pointer segments are of utmost importance, the chosen unit of abstraction is a linked segment (see section 3.3).

3.1 Bi-Abduction

This section gives an overview of what bi-abduction is, how it is used in combination with symbolic execution, and what role it plays in the analysis process in Broom and the join in particular. All explanations of bi-abduction below are based on the article on the topic by Calcagno et al.[CDOY11].

Bi-Abduction is a mixture of abductive analysis to synthesize preconditions and the frame problem, ie. finding postconditions or “leftover state”. Therefore, states in Broom consist of several parts:

- The **missing part**, containing the missing part of the precondition to be able to execute the code up to the current position, ie. the anti-frame; traditionally inferred using `abductive analysis`.
- The **current part**, containing the current state at a program point, ie. the frame; the question of the frame problem.

- A list of **existentially quantified variables** for the formulas above.

The bi-abduction problem is, given two formulas P and Q, to find the antiframe M and the frame F so that $P * M \models Q * F$ holds. Furthermore, it is desired for a bi-abduction procedure that solves this problem to find a solution that is as minimal as possible, and that especially does not lead to a trivially valid entailment (with an unsatisfiable formula on the left side).

Symbolic execution is applied to individual program instructions to generate pre- and postconditions for every statement (for example, a precondition not allowing a pointer to be null when it is dereferenced). For instruction I with precondition A_I (which needs to be fulfilled for the statement to be executed without errors) and postcondition B_I (guaranteed to be fulfilled after execution if the precondition was satisfied before), the contract $\{A_I\}I\{B_I\}$ (in Hoare triple notation, where $\{A\}P\{B\}$ denotes that assertions A and B are pre- and postconditions, respectively, for program segment P) for the instruction is added to the state via bi-abduction. For every function, the analysis starts with an empty state and adds equalities between program variables and their anchors (representing their value at the start of the function, no matter the current position). Then, the currently available interim contract $\{A\}P\{B\}$ (with the previously synthesized pre- and postconditions A and B for the program segment P) is taken and the bi-abduction problem $B * M \models A_I * F$ is solved to find a new antiframe M to add to the state's precondition, as well as a new frame F for the current postcondition. The new state $\{A'\}P; I\{B'\}$ then has $A' = A * M$ as its missing precondition and $B' = B * F$ as the current postcondition. This procedure is repeated and the state is updated throughout the function, finding contracts for nested blocks and function calls before their parents (which are then treated just like contracts for single instructions). For loops, fixed points need to be found to serve as contracts for the entire loop. After contracts for all instructions in a function have been added to a state, the result is a sound contract for the function itself and can be used for the analysis of functions calling it.

Because this combines both sides into one analysis step, contracts contain both a pre- and a postcondition after just a single run through the program. At the time of writing, a second run is required to verify that contracts hold for all branches of a function, including ones that were skipped during the first run. In this second run, preconditions of the first run are used to find postconditions again, and preconditions that might lead to errors in certain branches are discarded. However, there are efforts to change this and ensure the soundness of contracts after just a single run[SRVZ23].

The fact that bi-abduction is used, as opposed to finding pre- and postconditions separately, has a great influence on the requirements for the join operator. Most prominently, it means that both the missing and the current part must be joined simultaneously and that results must fit together. Otherwise, seemingly irrelevant names for logical variables now gain a lot of importance, and simple renaming is not possible anymore (whereas usually, giving an existential variable a new name results in an isomorphic formula). Additionally, removing seemingly redundant variables (equivalent to another variable) might in some cases not be possible anymore. The solution to this problem is discussed further in section 5.2.10.

3.2 Separation logic

Naturally, to analyze a certain aspect of a program, an expressive enough logic is required to describe resulting contracts. Since the main goal of Broom is to reason about heap-allocated objects, separation logic[Rey02, O'H19] seems like a reasonable choice. This section gives a short overview of separation logic, based on the paper of the same name by Reynolds[Rey02].

Separation logic, as an instance of the logic of bunched implications[OP99], is an extension to standard Hoare logic, defining additional operators and their semantics. Therefore, standard logical operations, such as conjunction, equivalence, or comparison, as well as arithmetical operations, continue to be available alongside the newly introduced operators. Separation logic includes the following additional operators:

- The **separating conjunction ***, or “star operator”, is similar to a normal conjunction operator; however, it additionally requires the two connected clauses to be in disjoint parts of memory. This way, reasoning about memory without the problem of aliasing is possible.
- The **points-to operator ->** describes a pointer as used in conventional programming languages, ie. a variable containing the memory address of a certain piece of data. This is the central part of separation logic that makes analysis of heap-allocated objects possible.
- The **separating implication -$*$**, or “magic-wand operator”, describes an implication in a certain heap, stating that if the heap is extended in some way, a certain attribute holds. This operator, however, quickly leads to undecidability[PZ22], especially in the presence of a list segment abstraction[HPR$^+$22], and, since it is not required to describe formulas generated by Broom, it is not further discussed in this thesis.

For the join operator, especially the presence of the points-to operator is a key factor for its functionality. Since all spatial predicates in Broom are connected via the separating conjunction, it is safe to treat this just like a conventional conjunction (while ignoring aliasing).

3.3 List abstraction

To make static analyses terminate, there always has to be some layer of abstraction between formulas in contracts and the real world. In Broom, the central entity in question is the linked list, and so new predicates are introduced to describe pointer segments of arbitrary length. This section first describes the structure of these predicates, followed by when and how they are generated by Broom[HPR+19].

3.3.1 Singly-linked segments

The simplest case for a linked list is a singly linked one, with each element containing a link to the next at a certain offset. To describe a list segment, a start, and an end are needed, as well as information as to how to proceed to the next element, given any element in the segment, together with any additional information about the respective element. In addition, Broom supports shared variables, ie. variables that point to the same memory location for all elements in a list. For example, all entries in a certain list might have a pointer back to the list start.

Therefore, a singly-linked list segment is constructed as follows:

Listing 3.1: Signature of a singly-linked list segment

```
Slseg(src, dest, lambda, shared)
```

Here, `src` is the start and `dest` is the end of the segment. `lambda` contains information about a single element (including the pointer to the next element) and `shared` is a list of expressions shared between elements, usually referencing variables in the containing formula.

A lambda consists of a list of two or more parameters (ie. variable names) and a formula. The first parameter always corresponds to the current element, while the second one describes the subsequent element. The formula then describes the current element via the first parameter, and how the next element can be reached from it (presumably through pointers originating from the current element). All remaining parameters directly correlate to the list of shared expressions (the third parameter to the first shared expression, and so on), so these variables being used in the lambda formula represent references to the respective expressions in the outer formula.

For example, take the lambda from Listing 3.2: The parameters `%l1, %l2`, and `%l5` correspond to the source, destination, and the first shared argument (`%l7+8`), respectively. Therefore, for each link in the segment, the pointer at offset 0 points to the next element in the segment, the pointer at offset 8 to `NULL`, and the pointer at offset 16 to some other memory address `%l7+8`, the same for every element in the segment.

No assumptions about the length of a segment are made; however, segments may not be empty and instead must contain at least one link.

3.3.2 Doubly-linked segments

In real-world examples, a commonly used structure is the doubly-linked list, extending a singly-linked one with a link back to the previous element, and thus allowing for traversal of a list in both directions.

Listing 3.2: Example lambda

```
lambda[%l1,%l2,%l5]:{
      %l1-(8)->%l2 * (%l1+8)-(8)->%l3 * (%l1+16)-(8)->%l5 &
      (%l1!=NULL) & (%l3=NULL)
},[%l7+8]
```

The predicate for doubly-linked segments in Broom looks similar to the one for singly-linked segments:

Listing 3.3: Signature of a doubly-linked list segment

```
Dlseg(fst, fst_backlink, lst, lst_fwdlink, lambda, shared)
```

`fst` and `lst` correspond to `src` and `dest` from the predicate for singly-linked segments, respectively, describing the two ends of the list. In addition, the two expressions `fst_backlink` and `lst_fwdlink` are captured, describing the forward link for the last and the backlink for the first element. This is necessary since, as opposed to singly-linked segments, where the target can be an arbitrary piece of data, here both the first and last element are list elements and therefore need to possess both of the pointers (or possibly pointer chains) to the successor elements in both directions. In many cases, these expressions will simply be null (indicating the end of a list); for cyclic lists, they will contain references to `lst` and `fst`, respectively.

The lambda for a doubly-linked segment follows a similar structure to one of a singly-linked one, with one major difference: Since obviously information about a backlink has to be present in addition to the forward link, another parameter (the third one) is introduced to represent the successor element in this direction. Parameters for shared expressions then start at position four.

3.4 Formula structure

Formulas in Broom are separated into two parts: Spatial predicates Σ and pure predicates Π. Spatial predicates include everything that describes the heap, namely points-to-predicates as well as all kinds of list segments. The pure part contains everything else, ie. predicates from standard Hoare logic. All predicates in the spatial part are joined together with the separating conjunction (the star operator *); in the pure part, conventional conjunction is sufficient. A formal grammar definition for Broom formulas can be found in Listing 3.4 (with trivial definitions, such as `int` or `string`, left out for brevity).

In formulas, all variables marked as anchors (`_anch` suffix) denote program variable anchors, ie. the value of program variables at the time when the function is called. As opposed to the variables themselves, this value can never change and can be used to reference initial parameter values in the postcondition.

Listing 3.4: Formal grammar of Broom formulas in EBNF

```
formula      = [spatial-part "&"] [pure-part]
spatial-part = spatial-pred {"*" spatial-pred}
pure-part    = expr {"&" expr}

spatial-pred = pointsto | slseg | dlseg
pointsto     = expr "-(" sof ")->" expr
sof          = expr
slseg        = "Slseg(" expr ", " expr ", "
                   lambda ", [" expr-list "])"
dlseg        = "Dlseg(" expr ", " expr ", "
                   expr ", " lambda ", [" expr-list "])"
lambda       = "lambda[" expr-list "]:{" formula "}"

expr-list    = [expr {", " expr}]
expr         = var | const | unop | binop | "Void" | "Undef"
var          = free-var | pvar | special-lvar
free-var     = "%l" int
pvar         = "%mF" int ":" (varname | anch_name)
varname      = a-zA-Z_ {a-zA-Z0-9_}
anch_name    = varname "_anch"
special-lvar = "%ret" | "%rF" int
const        = int | "true" | "false" | string | float
unop         = func | ("~" | "!" | "-") expr
func         = ("base" | "len" | "stack" | "static" |
                "freed" | "invalid") "(" expr ")"
binop        = expr ("=" | "!=" | "<" | "<=" | "&&" | "||" | "<>" |
                     "+" | "-" | "*" | "/" | "%" | "&" | "|" | "^" |
                     "<<" | ">>" | "<<<" | ">>>") expr
```

Listing 3.5: Example formula in Broom

```
%l1-(8)->%mF10:x * (%l1+8)-(8)->%l2 *
Slseg(%mF10:x_anch,%l1,lambda[%l1,%l3]:{
      %l1-(8)->%l3 * (%l1+8)-(8)->%l2 &
      (%l1!=NULL) & (%l2=NULL)
},[]) &
(%l1!=NULL) & (%l2=NULL)
```

Take, for example, the formula in Listing 3.5. This formula contains two points-to-predicates in line 1, each with a size of target field of 8 (the number in the brackets), from the logical variable 1 with offsets 0 and 8 to the program variable x (variable 10) and the logical variable 2, respectively. In lines 2 to 5, a singly-linked list segment from the anchor of x (variable -10, the initial value of the parameter x when the function is called) to the logical variable 1, with the specified lambda (with its own spatial and pure part), is defined. Together, lines 1 to 5 comprise the spatial part of the formula. Line 6 contains the pure part, consisting only of equalities and inequalities (but could also contain, for example, integer comparisons).

3.4.1 Abstraction

Since no information about what is a list segment is included in input programs (at least not in a formal way, though only possibly informally via variable names, for example), originally, the bi-abduction procedure can only generate simple pointers. Therefore, there needs to be a separate abstraction step to convert these pointers to list segment predicates where applicable. This section describes when and how this is done in Broom at the time of writing[HPR$^+$19], excluding any abstraction done in the join operator (described in section 5.2.8).

For linear programs, abstraction is not strictly necessary since there is no risk of exponential state explosion. Therefore, abstraction is especially important for programs with multiple paths, particularly loops, where a fixed point has to be found. Therefore, abstraction is tried on new states at join points, ie. points where multiple program paths converge. The effects of abstraction at these points are discussed in more detail in Section 3.6.

The abstraction procedure in Broom seems relatively natural: Intuitively, everything that “looks” like a list segment is collapsed into one. A pointer chain “looks like” a list segment if the following criteria are fulfilled:

- The chain has a length of two or more links.
- For all elements, any additional pointers are compatible, ie. they contain similar information.
- No program variables are lost when abstracting the pointer chain into a list segment.

Of course, this can in some cases yield false positives, but the fact that abstraction is only done at join points, where coming across such a segment is relatively probable, makes the success rate high.

Pointers to the same destination for all segment entries are added to the list of shared expressions, and a fresh variable is introduced as a new lambda parameter. Then, the formula for the new lambda is equal to all reachable predicates from the source variable (ie. the first variable in the pointer chain), cut off at its successor.

If all elements contain a pointer back to the previous element, the segment is assumed to be doubly-linked. Since this is not always known at the time of abstraction (e.g. if a list is first traversed in the forward direction, and only after that backward, in a separate loop), segments might in some cases be wrongly abstracted as singly linked.

3.5 Low-level memory operations

One important aspect of Broom is that it does not only handle cleanly structured memory access but also low-level pointer operations, as commonly used in C programs. This section gives an overview of what such operations might be and what problems this can pose for the analysis and the join algorithm.

Firstly, pointer arithmetic is an important topic when analyzing C code[Min06]. This is already handled relatively well in Broom, which operates on a per-field basis (instead of using whole objects), and therefore does not place a lot of additional restrictions on the join operator, except for the requirement to handle pointer offsets.

Another point to consider that can very well have an impact on the join algorithm, is the concept of block splitting: In C, traditionally all memory allocation happens via a call to `malloc` which returns a block of memory of user-defined length. This block is then often used for multiple different objects by simply splitting the memory region as needed. In reality, allocating for several objects at once is often encouraged, since the overhead of `malloc` is minimized to a single call[DDZ94, LC02]. In addition, memory blocks or parts thereof might be reused for entirely unrelated purposes after not being needed anymore for their initial cause.

This results in the sizes and types of pointer targets not being constant over time. For example, directly after the call to `malloc`, the returned pointer points to an n-byte memory region with undefined content. At a later point in time, though, the very same pointer might only point to a single integer, with the rest of the block used for other, unrelated data. Of course, the rest of the block can in theory still be reached from the original pointer via pointer arithmetic, but this might not be desired anymore.

Furthermore, this problem might not even only arise over time, but possibly also in a single state: In some cases, a certain memory region might have multiple interpretations at the same time[Min06]. For example, an entry of a linked list might be interpreted as the list head (with information about the list, ignoring any data specific to the list entry itself), or as an object that may or may not be part of a linked list (paying attention to the objects fields, but treating list metadata as arbitrary data). The concept of this example is often used in the form of intrusive lists, where list metadata is embedded in some region of the objects to be ignored by code operating on the objects themselves[HPR$^+$22]. Another example is a pointer to an object, which can also be interpreted as a pointer to the first field of said object.

One additional function introduced into Broom formulas to be able to handle such low-level operations is the base function, returning the address of the start of the memory block (such as a region of memory returned by `malloc`) a certain address is in. This can be used to enforce pointer arithmetic to work by equating bases of multiple variables, therefore ensuring them to be in the same block, which is required to jump between them by pointer addition (at least when using constant offsets).

3.6 Join points

A join point is a location in a program where multiple paths converge, commonly encountered at the end of a loop. Here, there might be some previous states (for example, from previous loop iterations) and one or more new states (from the current iteration). This section describes how states are handled for blocks and entire functions, at what point in time entailment is done, and how fixed points are found without the join operator. Naturally, the introduction of a join operator must change this process.

During the analysis of a function, states are kept in a so-called state table, mapping unique identifiers of basic blocks (linear blocks of code without any branches or loops) to lists of states, as well as a counter indicating how many times the entailment procedure to find a fixed point for the block (if it comprises a loop) has been called. When this counter reaches a certain limit (five calls by default), the process is canceled and no contract for the containing function is found.

Basic blocks are executed symbolically, and states are built from the instructions or adopted from previously executed nested blocks. At the end of a block, these states are added to the corresponding state table for the id of the block. Now, abstraction is performed for all new states (configurable to be executed at the end of every block, or only at the end of loops), abstracting pointer chains into segment predicates (see Section 3.4.1). Then, the new states to be added to the state table are filtered, dropping any states that are entailed by any of the old states (indicating that all the information from the state was already there in an earlier iteration). Now, any unchanged states are removed. In addition to the original state lists S_{old} and S_{new}, now $S_{add} := \{s \in S_{new}.\neg\exists s' \in S_{old}.s' \models s\}$ is available.

After this step, the process continues in the other direction: Any states previously present in the map that are now entailed by one of the remaining new states are pruned. This way, new states that simply contain more information than old ones replace them instead of accumulating redundant information, leaving states in $S_{keep} := \{s \in S_{old}.\neg\exists s' \in S_{add}.s' \models s\}$.

Lastly, any new states (S_{add}) receive a flag indicating that they have not yet stabilized, and further iterations might be necessary. The next time the block gets symbolically executed, states with this flag serve as the initial states the execution engine will build upon.

This algorithm has a good chance to terminate for loops iterating over linked lists. If a linked-list-like structure is detected, it will be abstracted to a list segment immediately after the end of the loop block. Then, even though this abstracted state cannot yet be pruned by entailment from any old states (that do not contain the list segment predicate), old states will likely be removed (with the list segment entailing a concrete version, like a list with a single link expressed using points-to predicates), and symbolic execution will continue on the abstracted state. Here, another pointer will be added (alongside additional information about the next link in the list), which will be collapsed into the existing segment during the next abstraction step. Now, this new state does not contain any more information than its predecessor, and the first entailment will hold, dropping the state. If there are no new states left, analysis of the loop is complete and one or more (in case of additional branches inside the loop, or base cases like an empty list not collapsed into the abstraction) states will be stored as the final contracts of the block.

This procedure is repeated for every block in a function (evaluating inner blocks before their parents, and using their contracts when executing parent blocks) until contracts for the entire function are available or a limit is reached. In case a function calls another function with no available contract, its execution is aborted immediately.

Note that functions are evaluated strictly from leaf to root in the call tree. This way, contracts for called functions, if found, are always available. Recursion is not supported by Broom.

CHAPTER 4

Joining States

This section describes the problem at hand, including hard requirements for a join algorithm, as well as factors that are nice to have or should be as optimal as possible, without the aspiration to absolute perfection. Then, some overview and discussion about how this may be achieved and how other projects try to solve similar problems are given.

4.1 Problem description

An algorithm should be introduced that, given two states for the same program point (although at possibly different paths to get there), returns a single state representing joint information from both inputs or the information that the two states are incompatible. Furthermore, this algorithm should be integrated with the existing pipeline of the analyzer.

There are a few requirements for this algorithm:

- The algorithm should be correct.
- A join should be refused if it were to lead to too big of a loss of information.
- The algorithm may use abstraction to better achieve its goal.
- Joining two states should try to lead towards fixed points for loops.

The first point seems evident but has to be mentioned nonetheless. Correctness in this scenario means that given a program segment P and two states correctly describing said segment (ie. the state's contract holds for the program segment) $S_1 = \{A_1\}P\{B_1\}$ and $S_2 = \{A_2\}P\{B_2\}$, the join result of the two states $S^{\bowtie} = S_1 \bowtie S_2 = \{A^{\bowtie}\}P\{B^{\bowtie}\}$ should be a consequence of the two inputs. For example, $\{A_1 \wedge A_2\}P\{B_1 \wedge B_2\}$ or

$\{A_1 \wedge A_2\}P\{B_1 \vee B_2\}$ are valid outputs, but $\{A_1 \vee A_2\}P\{B_1 \wedge B_2\}$ in general is not (since deriving both postconditions from only one of the preconditions is not sound).

Point number two is less clearly enforceable than the first one since losing "too much" information is always a matter of context. Joining without any loss of information is possible for two semantically equivalent states; otherwise, the algorithm usually has to generalize in some way to get a joined state. This could be circumvented by introducing disjunctions for the affected predicates and adding implications from the corresponding predicates in the missing part to ones present on only one side in the current part to the latter; however, this would in many cases greatly inflate the size and complexity of states. Therefore, an algorithm should be found that might in such cases drop information, but not too much so as not to prevent a reasonable result from being found. This should be verified using real-world examples to perform especially well for patterns commonly encountered there.

The third point states that abstraction during the join process is possible, as long as it is done correctly. This, however, is already verified by the first point, checking for entailment between the old, unabstracted version and the possibly abstracted result.

The last point states that, for two correct join results $S_1^{\bowtie}$ and $S_2^{\bowtie}$, the one entailing its inputs $(S_i^{\bowtie} \models S_1) \wedge (S_i^{\bowtie} \models S_2)$ is better. Oftentimes, however, such a result might not be possible, or multiple results fulfill this criterion.

Note that the terminology "join" here differs from what the word is usually used for when working with states generated by program analyzers: Instead of a join in a state lattice, which would lead to virtually unusable results (since lots of information would be lost there), the join operator needed is rather a widening operator that takes two states as its input and merges them. Nonetheless, the word "join" is used throughout this thesis for simplicity.

4.2 Approaches to joining

This section presents a few approaches to joining states, formulas, or similar structures. For each, advantages and disadvantages are outlined, and their relevancy for the joining of states in Broom is discussed.

4.2.1 The trivial join

Joining correctly is possible in every case in constant time by always emitting the trivial state $\{false\}P\{true\}$. This state is always valid for every program P and thus entailed by any input.

However, this state violates the second goal introduced in the last section: All information from the input states is lost, and so this state cannot possibly be a good join result (except for the case where both input states are empty as well).

4.2.2 Always refuse to join

There is another extreme, also trivial way to implement a correct join algorithm: No matter the inputs, always refuse to join. Termination can, at least in theory, still be reached if a state exactly (syntactically) matches a previously encountered state and the structure storing the states is a set that does not allow repeated elements. Obviously, in real-world examples, this will never occur, so just like always joining to a trivial state, this option does not work for analyzing real code.

4.2.3 Join by entailment

After two rather hypothetic variants of joining, this section presents a version usable for a variety of formula structures, even though the word “joining” is again a slightly questionable choice: Joining formulas by checking for entailment using a SAT solver such as Z3[DMB08] or cvc5[BBB^{+}22]. Here, when there is an entailment between two formulas (either $A \models B$ or $B \models A$), the formula on the right side of the entailment can simply be discarded (since all of the information is already present in the other formula), and the formula on the left can be taken as the join result. As both inputs to the call to join must be valid states, there is no need for the question of correctness as an already correct formula is simply inherited.

There is, however, one big condition for this approach to work: Entailment has to hold for subsequent iterations of a loop after a somewhat small set of initial ones; if, for example, a new pointer is added to a separation logic formula in every iteration and no abstraction is done to constrain the number of pointers, entailment can never hold for new states (due to the strictly increasing and thus never equal number of pointers), and the analysis will not terminate.

4.2.4 Opportunistic abstraction

To combat this problem, abstraction can be applied on a best-guess basis in hopes of making entailment possible. This is exactly what is done in Broom if no explicit join operator is present or enabled[HPR^{+}19]: Abstraction is applied wherever possible, and in a lot of cases this will lead to a more general formula that covers states from future iterations as well.

Of course, the success of this approach depends on whether changes to state in a loop can be abstracted in the given domain. Furthermore, this can lead to superfluous abstraction and thus in some cases worse (more abstract) results, even though a more concrete result could have been obtained with another strategy.

4.2.5 Disjunctive joining

The presumably most used option for merging multiple states into one is a relatively simple one; states can simply be concatenated with disjunctions[Cop14, HSS09, KKBC12, SHB16]. Combining a disjunctive list of states into one state disjunctively is trivially correct; then, depending on the structure of states, semantic-preserving operations can be applied to simplify the result. For example, for a state containing pre- and postcondition formulas, instead of merging on a state level, preconditions can be joined disjunctively if they are added as the left side of a newly introduced implication to the corresponding postcondition $((\{A_1\}P\{B_1\}) \vee (\{A_2\}P\{B_2\}) \Leftrightarrow \{A_1 \vee A_2\}P\{A_1 \rightarrow B_1 \wedge A_2 \rightarrow B_2\})$.

This works well in some cases of standard Hoare logic, where all clauses are pure and a SAT solver can relatively easily work with bigger formulas. However, since formulas joined like this get complex relatively quickly (containing most clauses from all input states), this can quickly become too much for a solver to handle[HSS09, KKBC12], especially if additional, more complex logic elements such as separation logic are added.

4.3 Joining states with heap predicates

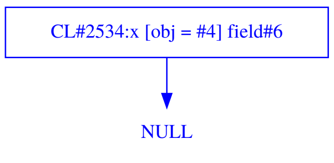
After a few more general approaches, this section focuses on approaches to join states for shape analyzers, ie. states that contain heap predicates. First, due to its close relation to Broom, Predator[DPV11, DPV14] is discussed in more detail; then, two more approaches are summarized.

4.3.1 Joining memory graphs

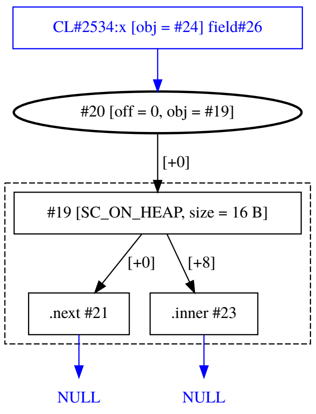
Broom is for many aspects inspired by Predator[DPV11, DPV14], another shape analyzer that, however, does not use separation logic but instead a graph representation of memory: Symbolic Memory Graphs. This section focuses on how joining states is done there and what role the join operation plays in the ecosystem; the information in this section stems mostly from the article on Predator's algorithmic details[DMP+13], as well as the implementation itself[DVP+16]. Illustrations for examples in this section are generated directly by Predator.

Symbolic memory graphs are graphs representing an abstract memory layout, with objects (single objects in memory or list segments) and values (single scalar values or pointers) as nodes and has-value (relating object fields to concrete values) and points-to (relating object fields to memory locations they point to) relations as edges. Singly- and doubly-linked list segments in Predator store a minimum length, making it possible to represent possibly empty (length 0+) list segments. When joining, the lower of the two minimum lengths is used. In addition, there are so-called 0/1 abstract objects, ie. objects that are either present or not (in which case pointers to them point to NULL instead). Shared expressions are made possible by storing a nesting level for all objects, indicating their scope.

The join operator in Predator serves several purposes: In addition to simply joining two states, it is the place where abstraction is done, and even entailment is done via joining by using the join status relation returned by the algorithm, indicating whether two graphs are equivalent, in an entailment relation, or incompatible.

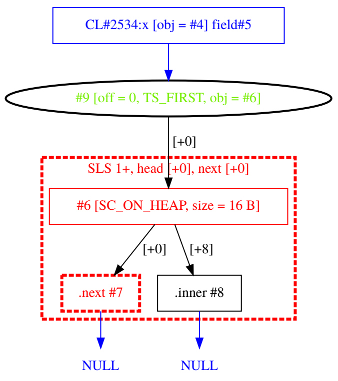


(a) Null variable

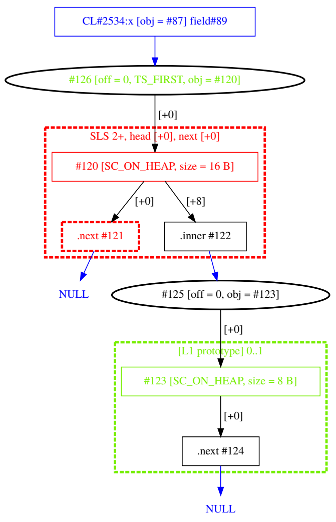


(b) Object

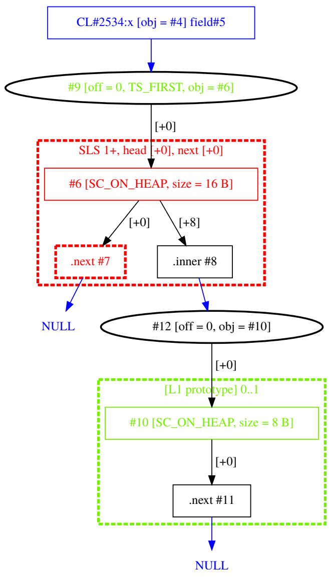
Figure 4.1: Null is never joined with an object


CL#2534:x [obj = #4] field#5
#9 [off = 0, TS_FIRST, obj = #6]
[+0]
SLS 1+, head [+0], next [+0]
#6 [SC_ON_HEAP, size = 16 B]
[+0]
[+8]
.next #7
.inner #8
NULL
NULL

(a) Null child


CL#2534:x [obj = #87] field#89
#126 [off = 0, TS_FIRST, obj = #120]
[+0]
SLS 2+, head [+0], next [+0]
#120 [SC_ON_HEAP, size = 16 B]
[+0]
[+8]
.next #121
.inner #122
NULL
#125 [off = 0, obj = #123]
[+0]
[L1 prototype] 0..1
#123 [SC_ON_HEAP, size = 8 B]
[+0]
.next #124
NULL

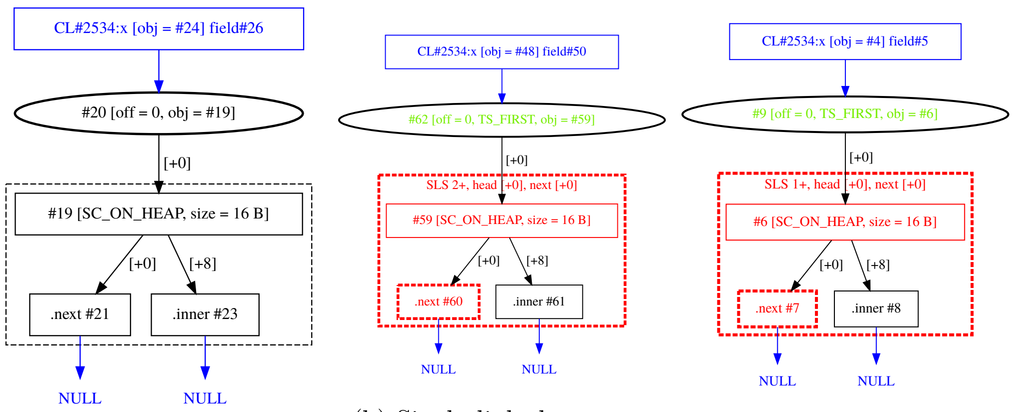
(b) 0/1 abstract object child


CL#2534:x [obj = #4] field#5
#9 [off = 0, TS_FIRST, obj = #6]
[+0]
SLS 1+, head [+0], next [+0]
#6 [SC_ON_HEAP, size = 16 B]
[+0]
[+8]
.next #7
.inner #8
NULL
#12 [off = 0, obj = #10]
[+0]
[L1 prototype] 0..1
#10 [SC_ON_HEAP, size = 8 B]
[+0]
.next #11
NULL

(c) Join output

Figure 4.2: 0/1 abstract objects can be joined with null

When joining two memory graphs, nodes are joined on a one-to-one basis; when two nodes are joined, a join reinterpretation function makes sure that their semantics match, or that the algorithm fails if they are too diverse. If joining two nodes is not possible, a list segment with length 0+ (which is possibly empty and can therefore occur practically everywhere) is introduced, and joining is tried again.

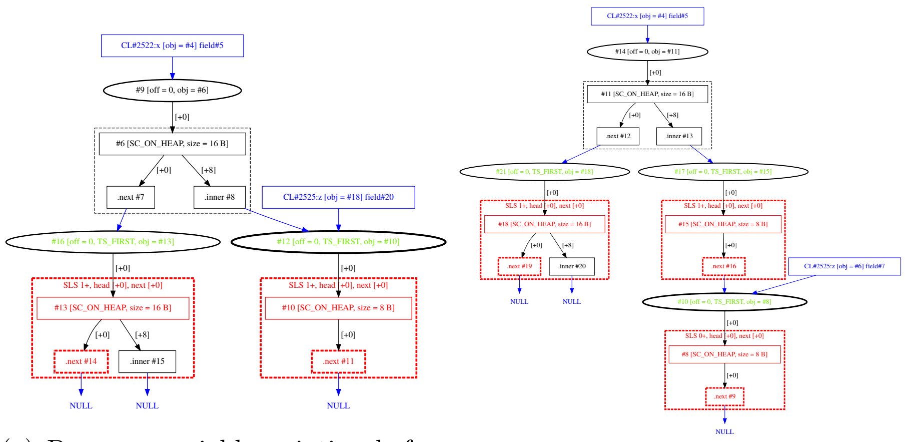

CL#2534:x [obj = #24] field#26
#20 [off = 0, obj = #19]
[+0]
#19 [SC_ON_HEAP, size = 16 B]
[+0]
[+8]
.next #21
.inner #23
NULL
NULL
CL#2534:x [obj = #48] field#50
#62 [off = 0, TS_FIRST, obj = #59]
[+0]
SLS 2+, head [+0], next [+0]
#59 [SC_ON_HEAP, size = 16 B]
[+0]
[+8]
.next #60
.inner #61
NULL
NULL
CL#2534:x [obj = #4] field#5
#9 [off = 0, TS_FIRST, obj = #6]
[+0]
SLS 1+, head [+0], next [+0]
#6 [SC_ON_HEAP, size = 16 B]
[+0]
[+8]
.next #7
.inner #8
NULL
NULL

(a) Object

(b) Singly-linked segment

(c) Join output

Figure 4.3: Objects can be collapsed into list segments



(a) Program variable pointing before segment

(b) Program variable pointing after segment

Figure 4.4: Program vars need to be preserved

The functionality of the actual join algorithm is relatively intuitive: Program variables are used as starting points, and objects are only joined if their children are compatible.

One interesting design choice in Predator is that NULL will never be joined with any object or non-empty list segments, but only with possibly empty segments or 0/1 abstract objects. This also means that the latter will never be generated by the join, but only by some other part of the system.

To illustrate the most important rules for joining in Predator, this paragraph presents some examples. Firstly, null cannot be joined with a simple object (figure 4.1), but joining is possible with 0/1 abstract objects (figure 4.2). Objects can be collapsed into list segments (figure 4.3), which changes their minimum length to 1. If necessary, such a segment will be created on the fly. Finally, all program variables are used as anchors that need to match in the output, so the join in Figure 4.4 is not successful (mind the variable z, which points to two different locations in the two input graphs).

4.3.2 Joining separation logic formulas

There have also been some attempts to join formulas with list predicates; this section presents two examples where this has been done.

The first example is presented in a paper by Yang et al.[YLB+08]. Here, the authors try to, given two input states, find a joined heap along with a list of tuples of variables (one from each side, plus a new variable representing the two in the joined state), as well as two lists of list segment start- and endpoints that have been generalized during join to now possibly empty segments, one for each input side. This is achieved by using a set of rules, searching for ones that match the input and satisfy any additional conditions. Input states are continually shrunk until only an empty state is left and the output contains the join result.

In addition to the output state, there are two additional entities in the algorithm's output, which serve an important purpose: Checking whether too much information has been lost during the join. This is accomplished by ensuring that any equalities now possibly present (due to possibly empty segments or because of joined expressions) are backed by both input states; if this is not the case, the join fails.

An example from the paper is joining (ls NE x 0 * $y \to 0$) and ($x \to x'$ * ls NE y x' * ls NE x' 0) (notations as from the paper, where ls denotes a list segment that is not empty (NE) or possibly empty (PE) between the other two parameters, for simplicity without any additional information other than the next pointer). When the join rules are applied to these two states, it will result in the state (ls NE x v' * ls NE y v' * ls PE v' 0), along with the variable join tuple $(0, x', v')$ (v' corresponds to 0 in the left input state, and x' in the right input state) and the generalization lists $\emptyset$ (no lists made empty on the left side) and $\{(x', 0)\}$ (there used to be a non-empty segment between x' and 0 in the right input state, but this is now possibly empty).

A second approach is presented by Li et al.[LBCR17]. Here, in addition to joining two states, instructions on how to choose which states to join are given: Each state's silhouette, ie. sets of vertices (variables) and paths (pointer chains) represented by regular expressions, is determined. Then, states are grouped by similar silhouettes, and regular expressions are generalized through a generalization function. States are finally joined by matching silhouette edges and joining the corresponding regular expressions by using the smallest and most general regular expression to represent the path.

For example, when analyzing a binary tree search algorithm, two silhouettes found could be $\{(t, l \cdot r \cdot l, x)\}$ and $\{(t, l \cdot l \cdot r, x)\}$, indicating paths through the left/right/left and left/left/right element, respectively, from tree root t to leaf x. These regular expressions can both be generalized to form the silhouette $\{(t, (l + r)*, x)\}$ ("any path through the left or right elements"). Since this silhouette is compatible with both inputs and no additional information is lost, the join can succeed.

CHAPTER 5

The Join Algorithm

The join algorithm implemented in the course of this thesis takes two sets of states as its input and then tries to perform pairwise joins between members of the two. The output it produces consists of two sets as well: States from the first input set that were not joined (ie. that should be kept as-is), and a union between join results and unjoined states from the second input set. The reason why join results are combined with unjoined states from the second input set is that this set is assumed to contain “new” states, e.g. states added after the latest iteration of a loop. If the need arises, this could easily be changed to e.g. return three separate sets; however, this was not necessary for this project. This chapter explains the details of the algorithm, with a formal definition of the join function.

5.1 Selection of pairs to join

Naturally, the input to the join algorithm consists of lists of formulas instead of just two single formulas, since there can be many distinct states at a given program point. Therefore, this poses the question of which of the formulas to join, independent of how they are joined.

The simplest approach to this problem is to just try joining formulas until a successful join is encountered and then continue the process with all formulas not joined before, ie. a brute-force approach. Since formulas will only ever be joined if they are compatible and thus similar enough for the join to make sense, this will usually yield a relatively solid result. One huge downside of this approach is that many “obviously” impossible joins will be tried anyway and the join algorithm will be executed possibly many more times than technically necessary. Furthermore, some states might be better off being joined with some particular other state, but are out of the pool after being joined with another, “inferior” state; however, this has proven not to be a deal-breaking problem in practice.

As an alternative to the brute-force version, one might try to check for compatibility before the join happens[LBCR17]. To do this, one might extract types (see section 5.2.1) for all input states and arrange them into “compatibility classes” (ie. sets of formulas possibly compatible with each other), later only joining inside the respective classes. This would decrease the number of unnecessary calls to join and could be implemented in a way where valid joins are never excluded (e.g. putting two states where a particular variable is scalar in the first and a pointer in the second state in two different compatibility classes cannot prevent a successful join from happening, since these variables could not be joined anyway). However, by keeping it this vague, the advantage over trying all possibilities is relatively small and easily outweighed by the burden of additional complexity, especially since such obviously impossible joins will be caught by the join algorithm soon anyhow, which will then immediately terminate. Therefore, this approach has not been implemented in the course of this project, and the focus was shifted towards an efficient join algorithm instead.

5.2 Joining two formulas

After two join candidates have been selected, the actual join process can start. Here, we will receive two states (referred to as “left” and “right” state here), each consisting of a missing and a current formula together with a list of existential variables, as an input, and return either a state that represents the result of joining the two input states, or a special “not joinable” token that indicates incompatibility of the two states.

As mentioned in Section 3.1, caution has to be exercised so as not to lose any relations between variables in the missing and current part. There are two approaches to achieve this: Either the two parts have to be joined simultaneously, or the algorithm has to make sure that no variables are renamed or removed that are part of such a connection. This implementation uses the latter approach; therefore, the two missing parts are joined separately from the two current parts, and all operations during the join preserve original variable names where applicable. Of course, there are cases where this is not entirely possible (e.g. when joining two differently-named variables); in this case, a check whether a variable that is still present on the other side (missing and current) of the final result was lost is performed, and if so a corresponding equality clause to the variable it was joined with is introduced.

The join algorithm has three steps: First, the input data is preprocessed and converted to a more suitable form, after which a new formula is synthesized (ie. the join is performed), which is then converted back to a state. These steps are described in detail in this section.

5.2.1 Preprocessing

Before two formulas can be joined, some normalization has to be performed to simplify the upcoming process. First, all logical variables (ie. existential variables) in the right state that occur also in the left state are replaced with fresh variables; this way, referring to such variables is unambiguous and two of the same variable always have to be joined (which, of course, is not the case for logical variables that coincidentally have the same name but might represent entirely different entities). For other variables (program variables and special logical variables like the return value) this is not done, since these certainly correspond to the same piece of memory (although after a different path through the program) and thus have to be joined.

After this first normalizing step, the states (or rather the missing and current formulas in the states) are converted to a format more suitable for joining, which we will refer to as `types`.

Types

Since the join algorithm operates on a per-variable basis, the idea of this transformation is to obtain a map between variable names and all the information available about the corresponding variable, ie. its `type`. Note, however, that this uses a far more abstract notion of a type than, for example, most modern programming languages, where types are often tightly coupled with concrete code that defines interfaces and behavior (e.g. classes in object-oriented languages). In this context, however, types only try to describe what is known about a variable in a specific state, without any regard for capabilities or changes during the program execution. It is also important to note that no type information is extracted from source code by Broom since assumptions made by the type system in C can easily be broken via type casts or pointer arithmetics; therefore, "type" information only comes from actual memory access patterns (concretized in formulas). At the time of writing, variable types fall into the following categories (which may be extended at a later point):

- **Variable types** with a variable name `v` and an offset `o` are references to the type of another variable `%v+o`.
- **Pointer types** represent pointers to memory regions and contain information known about these regions. This information consists of a map between known, constant offsets to field sizes (in bytes) and the types of said fields.
- The **null type** is used for variables that contain a pointer to null.
- **Scalar types** together with an optional concrete value represent variables containing an integer.
- Similarly, **boolean types** describe `true`, `false`, or an unknown boolean value.
- **Singly-linked segment types** contain, much like their counterpart in formulas, a lambda (which is another map from variable names to types) together with a source and destination variable, as well as a target variable and offset (ie. the end of the segment). Note, however, that shared expressions are not directly represented in slsegs but are instead moved to outer types.

- **Doubly-linked segment types** have not been implemented as part of the project; however, this would be a relatively trivial task: Most aspects are similar to singly-linked segments, except for the fact that an additional backward destination variable exists for its lambda. Note, however, that this type has to be inserted not only for the start variable but also for the "end" of the segment, which can in reality be seen as the start of the inverse doubly-linked segment (with next and previous destinations flipped). Broom formulas also include "back pointer from first" and "forward pointer from last" fields, but these can just be inserted into the pointer type for the respective variables.

- **Outer types** are references to variables one step out of a lambda to its parent type map, isomorphic to shared expressions for lambdas. For example, in a type map with an slseg that in turn contains an outer type to variable `%x+0`, this indicates a reference to `%x` in the original map instead of the lambda (the "global" `%x` if the segment is not nested), as opposed to the variable type `%x+0` that references the variable `%x` in the lambda. As a Broom formula, this variable corresponds to a lambda parameter with `%x` as its value. In nested list segments, outer types point into their next parent segment. An outer type in the topmost map is meaningless and will never be generated.

- **Wildcard types** are used for variables that can be joined with anything; these will not be generated from formulas, but prove to be useful as intermediate types when joining lambdas.

- Lastly, **unknown types** are for variables that are known to exist, but where no further information is given.

A hierarchical overview of all available types can be seen in Figure 5.1.

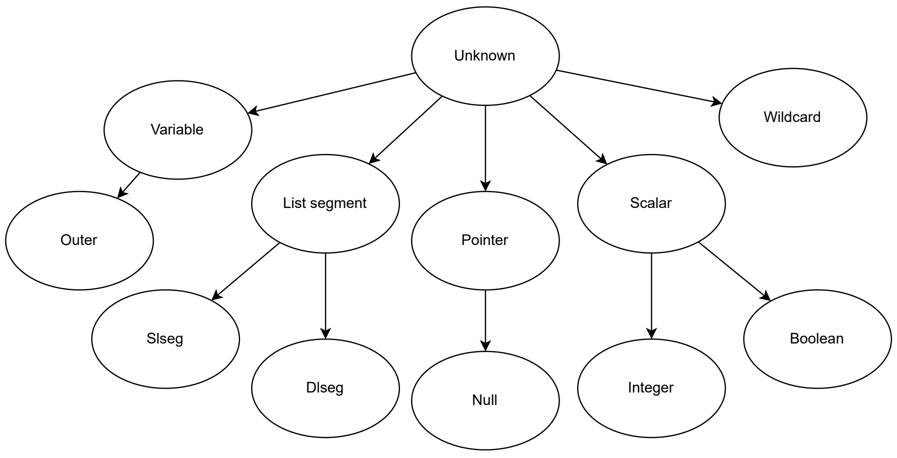


Figure 5.1: Type hierarchy for join types

This set of types, however, is not enough to represent all possible formulas. Therefore, type maps are augmented with **constraints** where applicable:

- **Program variable constraints** indicate that a variable is a program- or special logical variable.
- **Alias constraints** are generated when a variable is equivalent to another variable with an optional constant offset.
- **Null constraints** explicitly state that a value is null also as a constraint (which is important since constraints are merged separately).
- **Not-null constraints** are for values that are explicitly not equivalent to null.
- **Constant constraints**, just like null constraints, explicitly state that a variable has a known, constant value.
- **Not-equal constraints** represent inequalities to constant values.
- **Base constraints** are for the common pattern in Broom to equate the bases (addresses of the start of the containing allocated memory block, e.g. using `malloc`) of two variables. They contain two variables with offsets, at least one of which should be equivalent to the variable it constrains.
- Finally, **expression constraints** are used for any expressions not explicitly handled by the join algorithm and contain the original expression from the input formula, as well as the variable name the expression was initially for (to enable variable renaming).

Generating a type map from a formula is a very natural process:

- Each variable starts as an unknown type with an empty set of constraints.
- Equivalences (to null, numbers, or booleans) are converted to the corresponding type. For example, `%x = 4` would give `x` the type `Scalar(4)` and the constraint `Const(4)` (as well as a `PVar` constraint since `x` is a program variable).
- Equivalences to other variables with a constant offset are converted to alias constraints, and another conversion run is done with the corresponding variable replaced with the current one. For instance, `%l2 = %x+8` would result in the constraints `Alias(x, 8)` for `%l2` and `Alias(l2, -8)` for `x`.
- Inequalities (not equal, less, and less than or equal) are converted to constraints, and the type is fixated to match (e.g. for inequalities with integers, a scalar integer type is assumed). Therefore, `%x != 4` results in the type `Scalar(?)` and constraint `Neq(4)` for `x`.

Listing 5.1: Example type map for pointer type

```
%x: {
  0-(8)->Slseg(%l5, lambda(%l1, %l2) {
    ...
  }),
  8-(8)->%l1
}
```

- Equivalences between two calls to the `base` function are converted to base constraints. For the predicate `base(%l1) = base(%l2+8)`, this would result in the constraint `Base(l1, 0, l2, 8)` for both variables.
- All other pure expressions, including expressions with complex calculations, are converted to expression constraints.
- Any variable that is involved in a spatial predicate as a source (pointer or list segment) has a pointer type; for pointers, the corresponding value at the given offset in the pointer type map is a variable type, for list segments, it is a list segment type. In lambdas, additional parameters (after source and destination) are given outer types. For example, the input formula `%x+8-(8)->%l1 * Slseg(%x, %l5, lambda[%l1, %l2]:{...}` results in the type seen in Listing 5.1.
- When additional spatial predicates are encountered, the pointer type is updated and the corresponding pointer or list segment is added.
- Types are only ever updated and made more concrete, never replaced (except for the unknown type). If inconsistencies are encountered (e.g. a variable is both a pointer and a scalar), the join immediately fails.

In the above conversion rules, expressions are normalized (e.g. converting subtractions to additions with a negated second parameter or swapping operands of additions to move variables before constants), and offsets are found by matching common expression patterns. This certainly does not find constant offsets in every expression it possibly could, but these simple rules have in practice proven to be enough to handle most formulas generated by Broom.

A more comprehensive example is presented for the formula in Listing 5.2; this would be converted to the type map in Listing 5.3, with constraints appended in brackets. Figure 5.2 shows this type map as a graph (constraints omitted for brevity) for easier visualization: x and $l1$ are pointers, $l2$ is null, $l4$ is scalar, the top box represents the list segment from $l1$ to $l3$ with an outer variable $l7$ (note that this has its own variable scope, hence the duplicate variable names), and $\%ret$ is unknown; edges are labeled with the sizes of the respective fields.

Listing 5.2: Example input formula

```
%mF10:x-(8)->%l3 * (%mF10:x+8)-(8)->%l1 *
(%mF10:x+16)-(4)->%l4 * %l1-(8)->%l2 *
Slseg(%l3,%ret,lambda[%l1,%l3,%l7]:{
      %l1-(8)->%l3 * (%l1+8)-(8)->%l2 * (%l1+16)-(4)->%l7 &
      (%l1!=NULL) & (%l2=NULL) &
      (base(%l1)=base(%l1+8)) & (base(%l1)=base(%l1+16))
},[%l4]) &
(%mF10:x!=NULL) & (%l1!=NULL) & (%l2=NULL) & (%l4=17) &
(base(%mF10:x)=base(%mF10:x+8)) & (base(%mF10:x)=base(%mF10:x+16))
```

Listing 5.3: Example output type map

```
%x: {
  0 -(8)->%l3,
  8 -(8)->%l1,
  16-(4)->%l4
} (NotNull, PVar, Base(%x, 0, %x, 8), Base(%x, 0, %x, 16))
%l1: {
  0-(8)->%l2
} (NotNull)
%l2: NULL (Null)
%l3: {
  0-(8)->Slseg(%ret, lambda(%l1, %l3) {
   %l1: {
     0 -(8)->%l3,
     8 -(8)->%l2,
     16-(4)->%l7
   } (NotNull, Base(%l1, 0, %l1, 8), Base(%l1, 0, %l1, 16))
   %l2: NULL (Null)
   %l3: *
   %l7: Outer(%l4)
  })
}
%l4: Scalar(17) (=17)
%ret: Unknown (PVar)
```

After types have been generated, the left and right type maps for the missing part and the two maps for the current part are joined into one map each, augmenting the key (previously consisting only of variable names) with a `side`, Left or Right, to indicate where a type comes from and the fact that it has not been joined yet. The third side, Both, is used later for joined variables.

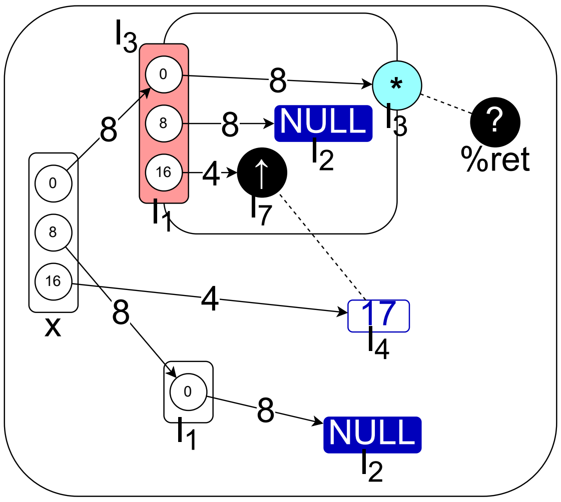


Figure 5.2: Graph representation of example type map

When these steps are completed, the two maps (missing and current) are ready for the join algorithm itself, which will be applied to the two individually.

5.2.2 Joining Types and Constraints

After preprocessing, the task of the join algorithm is to make sure all relevant variables (ones reachable from program variables) have "Both" as their side and to update their contents accordingly. Therefore, the join operates on a per-variable basis, starting with all available non-free variables, favoring anchor variables (which are not mutable) over program variables or special logical variables such as the function return value.

In the following sections, a function $\bowtie (tm, t_1, t_2) \mapsto (tm', res)$, $res \in \{\text{Type } t, \text{ Incompatible, TryAbstraction}\}$ will be introduced that joins the two types t_1 and t_2 in the context of the type map tm and returns a new type map tm', as well as a join result res. Here, $tm[v, s]$ retrieves the type for variable v and side $s \in \{L, R, B\}$ from the type map tm (or Unknown if not present), $ptr[o]$ retrieves the pointer member at offset o from pointer ptr, $tm\{a \mapsto b\}$ updates type map tm with type b at position a, $\mathit{offset}(t, o)$ offsets type t by o bytes, $\mathit{parent}(tm)$ returns the parent type map of tm (the type containing the respective list segment), $\mathit{outer}(t)$ converts outer type $\mathit{Outer}(v, o)$ to variable type $Var(v, o)$, $pg(p_1, p_2)$ groups the two pointer arguments as described in section 5.2.5, $\mathit{reachable}(tm, ptr)$ filters type map tm for all members reachable from ptr, and $\mathit{offsets}(ptr)$ gets all defined offsets of pointer ptr.

To accomplish this, variable types for program variables and their anchors are "resolved" one after another, requesting a single (joined) truth for each of them, as well as any reachable variables. During all steps for type resolution, the type map is updated accordingly, ending in a completely joined map if possible. This process can be seen in Algorithm 1.

Algorithm 1 The resolve function

```
function RESOLVE(type, map)
    if type is Var(var, offset) then
        if (var, Both) $\in$ map then
            return map[var, Both]+offset, map
        else
            left $\leftarrow$ map[var, Left]+offset
            right $\leftarrow$ map[var, Right]+offset
            return JOIN(left, right, map)
        end if
    else
        return type, map
    end if
end function
```

For example, look at the type graph in Figure 5.3: Initially, all program variables, in this case `x` and `y`, are requested and joined if still one-sided (green dashed lines). These joins can, as is the case for `y` in this example, trigger additional joins (blue dashed lines): l_1 from the left and l_5 from the right side are joined (despite their differing variable names), and the same happens for the two sides of l_2. Cases where this happens are discussed in the following sections, with this concrete example handled in section 5.2.5. Variables not reachable in this way, like l_3 in the example, are not relevant for the output and thus not joined, even if they are present on both sides. How each of these joins is performed is explained starting in Section 5.2.3.

During a resolve or join operation, valid results are `Incompatible`, a `Type` containing a concrete, new type, `TryAbstraction` to indicate an impossible join unless abstraction is applied, or `Compatible` if the two types are interchangeable (only applicable for a join operation, not for resolve operations).

$$
\begin{aligned}
&resolve(tm, Var(v,o)) && && (5.1)\\
&\qquad = tm, \mathit{offset}(tm[v,\mathrm{B}], o) && |(v,\mathrm{B}) \in tm && (5.2)\\
&\qquad = tm'\{v \mapsto t'\}, t && |l = \mathit{offset}(tm[v,\mathrm{L}], o), r = \mathit{offset}(tm[v,\mathrm{R}], o), && (5.3)\\
& && |tm', t = \bowtie (tm, l, r), t' = \mathit{offset}(t, -o) && (5.4)\\
&resolve(tm, t) = tm, t && |\text{for other types} && (5.5)
\end{aligned}
$$

When a type resolution is requested, there are multiple possible cases, as defined in the equations at 5.1:

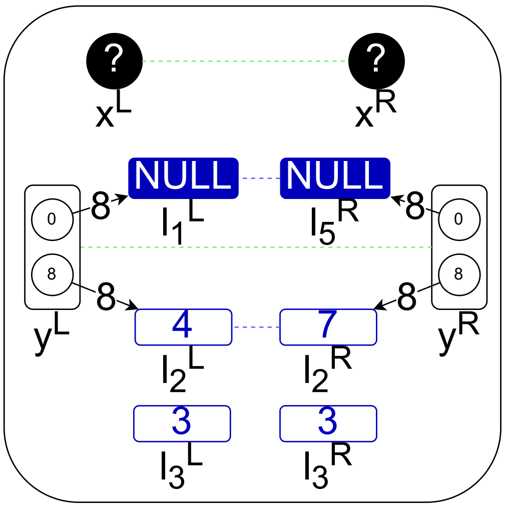


Figure 5.3: Example type map joining left and right sides

- **The type is not a variable type**: In this case, the type itself is returned and it is expected to be joined by the calling context.
- **The type is a variable type which has side `Both` in the type map**: The type from the type map is returned.
- **The type is a variable type but has not yet been joined (either `Left`, `Right`, or both variants are present in the type map)**: All available variants (up to two) are retrieved and passed to the type join function (see Section 5.2.3).

5.2.3 Joining a single type pair

Algorithm 2 shows the top-level orchestrator function for joining two types in the context of a type map. When joining two types, there are first a few base cases to consider:

- If the types are variable types that have already been joined, return `Compatible`.
- If exactly one of the two is an outer type, fail.
- If both of the types are outer types, continue joining in the parent context (or fail if no such context exists).

After these cases have been checked, the types are resolved (which, as previously stated, does nothing if the types are already concrete, but makes sure that beyond this point there are no more variable types - which, if present, are joined in another call). If this fails, the failure is passed on to the calling context.

Algorithm 2 The top-level join function

```
function Join(left, right, map)
    if left is Var and right is Var and (left, right) are joined then
        return Compatible, map
    else if (left is Outer) $\neq$ (right is Outer) then
        return Incompatible
    else if left is Outer($v_l, o_l$) and right is Outer($v_r, o_r$) then
        return Join(Var($v_l, o_l$), Var($v_r, o_r$), parent(map))
    end if
    left, map $\leftarrow$ Resolve(left, map)
    right, map $\leftarrow$ Resolve(right, map)
    constraintResult $\leftarrow$ JoinConstraints(left.constraints, right.constraints)
    if constraintResult = $\perp$ then return Incompatible
    end if
    if left is * then return right, map
    else if right is * then return left, map
    else if left is Lseg or right is Lseg then
        result, resultMap $\leftarrow$ JoinLseg(left, right, map)
    else if left is Ptr and right is Ptr then
        result, resultMap $\leftarrow$ JoinPtrs(left, right, map)
    else
        result, resultMap $\leftarrow$ JoinScalar(left, right, map)
    end if
    for all alias $\in$ aliases(left) $\cup$ aliases(right) do
        result, resultMap $\leftarrow$ Join(result, alias, resultMap)
    end for
    result.constraints $\leftarrow$ constraintResult
    update left and right to result in resultMap
    return result, resultMap
end function
```

After resolving types, constraints are joined (see Section 5.2.7). If this is successful, types are checked for their kind:

- If one of the two types is a wildcard type, the other type is kept literally.
- If one or both types are list segments, the algorithm to join list segments (see Section 5.2.6) is applied.
- If both types are pointer types, their children are joined (see Section 5.2.5).
- Otherwise, at least one of the types is not spatial, and the types are thus joined by the algorithm for joining scalar types (see Section 5.2.4).

Table 5.1: Join matrix for kinds of types, with references to sections explaining the join where applicable

$t_1 \backslash t_2$	*****	**?**	**Outer**	**LSeg**	**Ptr**	**NULL**	**Num(n)**	**Bool(b)**
*****	$*$	t_2	$\bot$	t_2	t_2	t_2	t_2	t_2
?	t_1	?	$\bot$	$\bot$	abstract	NULL	Num(?)	Bool(?)
Outer	$\bot$	$\bot$	cont. in parent	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
LSeg	t_1	$\bot$	$\bot$	Sec. 5.2.6	Sec. 5.2.6	$\bot$	$\bot$	$\bot$
Ptr	t_1	abstract	$\bot$	Sec. 5.2.6	Sec. 5.2.5	$t_1 = \emptyset \rightarrow t_1$	$\bot$	$\bot$
NULL	t_1	NULL	$\bot$	$\bot$	$t_2 = \emptyset \rightarrow t_2$	NULL	$\bot$	$\bot$
Num(m)	t_1	Num(?)	$\bot$	$\bot$	$\bot$	$\bot$	Num(n=m) Num(?)	$\bot$
Bool(a)	t_1	Bool(?)	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	Bool(a=b) Bool(?)

Keep in mind that it is not possible to have variable types here, as they were resolved in a previous step.

Table 5.1 shows a matrix that gives an overview of how certain kinds of types are joined.

After a type is joined, all of its aliases (found via alias constraints) are updated. If any of these have not been joined yet (ie. they have `Left` or `Right` as their side), the whole join algorithm is repeated with the result of the current join and the type(s) of the variable in question to account for alias chains alternating between left and right input variables.

Formally, these rules can be seen in the equations starting at 5.6.

$$\bowtie(tm, t_1 = Var(v, o), t_2) \qquad \text{(symmetrical for } t_2\text{)} \tag{5.6}$$
$$= \bowtie(tm', t_1', t_2) \qquad | tm', t_1' = resolve(tm, t_1) \tag{5.7}$$
$$\bowtie(tm, t_1, t_2) \tag{5.8}$$
$$= \text{Incompatible} \qquad |\text{exactly one of } t_1, t_2 \text{ is outer} \tag{5.9}$$
$$= \bowtie(parent(tm), outer(t_1), outer(t_2)) \qquad |\text{both } t_1 \text{ and } t_2 \text{ are outer} \tag{5.10}$$
$$\bowtie(tm, *, t_2) \qquad \text{(symmetrical)} \tag{5.11}$$
$$= t_2 \tag{5.12}$$

5.2.4 Joining scalars

Joining scalar types (including Null and unknown types) follows a basic set of mostly intuitive rules, formalized in the equations consecutive to 5.13 and Algorithm 3:

Algorithm 3 Function for joining scalars

```
function JoinScalar(left, right, map)
    if left is Unknown then
        if right is Ptr then return TryAbstraction(LEFT)
        else return removeValue(right), map
        end if
    else if right is Unknown then
        if left is Ptr then return TryAbstraction(RIGHT)
        else return removeValue(left), map
        end if
    else if left = right is NULL then return NULL, map
    else if {left, right} = {NULL, Ptr(∅)} or vice versa then
        return Ptr(∅), map
    else if left is Num(l) and right is Num(r) then
        if l = r then return Num(l), map
        else return Num(?), map
        end if
    else if left is Bool(l) and right is Bool(r) then
        if l = r then return Bool(l), map
        else return Bool(?), map
        end if
    else return Incompatible
    end if
end function
```

- When joining Unknown, it is assumed that the other side carries the truth about the type, and so it is simply carried over. For scalar types that have a concrete value (number or boolean), this value is removed and an "unknown scalar" type is returned. When joining Unknown with a pointer, the premature end of a list segment is assumed, and abstraction is requested.
- Null types can only be joined with Null and empty pointer types (which do not have a direct representation in formulas and are basically "something that is not null"), as well as unknown and wildcard types. If a list segment is to be joined with a pointer to null, its target is joined with null instead (indicating a one-step segment to null).
- Scalar (number or boolean) types can be joined with other scalar (number or boolean, respectively) types, but lose their concrete value if it is not equivalent. All other types (except for unknown) lead to an incompatible join.

This information is also included in Table 5.1. Note that these rules are not necessarily correct for the missing side of a state - for this problem, see Section 5.3.

$$\bowtie(tm, Unknown, Scalar(v)) \quad \text{(symmetrical)} \tag{5.13}$$
$$= tm, Scalar(?) \quad |\text{ for int and boolean scalars} \tag{5.14}$$
$$\bowtie(tm, Unknown, Ptr(p)) \quad \text{(symmetrical)} \tag{5.15}$$
$$= tm, TryAbstraction \quad |p \neq \emptyset \tag{5.16}$$
$$\bowtie(tm, Unknown, t_2) \quad \text{(symmetrical)} \tag{5.17}$$
$$= tm, t_2 \tag{5.18}$$
$$\bowtie(tm, Null, Null) \tag{5.19}$$
$$= tm, Null \tag{5.20}$$
$$\bowtie(tm, Null, Ptr(\emptyset)) \quad \text{(symmetrical)} \tag{5.21}$$
$$= tm, Ptr(\emptyset) \tag{5.22}$$
$$\bowtie(tm, Null, Slseg(t, \lambda)) \quad \text{(symmetrical)} \tag{5.23}$$
$$= tm', Slseg(t', \lambda) \quad | \bowtie (tm, Null, t) = (tm', t') \tag{5.24}$$
$$\bowtie(tm, Scalar(v_1), Scalar(v_2)) \quad \text{for int and boolean scalars (same type)} \tag{5.25}$$
$$= tm, Scalar(v_1) \quad |v_1 = v_2 \tag{5.26}$$
$$= tm, Scalar(?) \quad |\text{otherwise} \tag{5.27}$$

5.2.5 Joining pointers

Joining pointers, as shown in Algorithm 4, is where the algorithm gets recursive. First, types at offsets available on both sides are joined pairwise, and the sizes of the relevant fields are joined (which should be straightforward, since they can only be constant or variables with offsets, and never more complicated types). If two field sizes cannot be unambiguously joined (e.g. because a block was split between the two states and the field was shortened), the join fails, since this can only happen a finite amount of times (no block can be split infinitely often, so chances are high that the analysis will still converge) and there is nothing to join the leftover memory section (from the now shorter region) with (since there is no pointer on the side where the section is an unsplit single block). Then, the parent variables of the pointer types are marked as already joined for pointer targets to be joined pairwise to avoid endless recursion in case the type references itself. This can safely be assumed since either there is something else preventing a join in which case the join will fail anyway, or the join will succeed in which case the assumption that the variables can be joined is true.

After all relevant offsets have been joined, results are checked for incompatibilities (in which case this join fails as well) or cases where a join is only possible if abstraction is applied. In the latter case, abstraction is tried as described in section 5.2.8.

If all child joins are compatible (possibly after abstraction has been done and the join repeated), keys on both sides are compared, paying regard to offsets possibly "hidden" in list segments (for example, if a list segment starting at `$x+8` defines a pointer from its

Algorithm 4 Function for joining pointer types

```
function JoinPtrs(left, right, map)
    if offsets do not match then
        if left = $\emptyset$ or right = $\emptyset$ then return TryAbstraction(whichever side is $\emptyset$)
        else return Incompatible
        end if
    end if
    mark variables for left and right as Joined
    result $\leftarrow$ Ptr($\emptyset$)
    for all offset, group $\in$ GroupOffsets(left, right)
        if Lseg $\in$ group then
            innerResult, map $\leftarrow$ JoinLseg(group.left, group.right, map)
        else
            innerResult, map $\leftarrow$ Join(group.left[offset], group.right[offset], map)
        end if
        result[offset] $\leftarrow$ innerResult
    end for
    if TryAbstraction(side) $\in$ result then
        abstracted $\leftarrow$ Abstract(side ? left : right, map, offset in result)
        return Join(abstracted, side ? right : left, map)
    else if Incompatible $\in$ result then return Incompatible
    else return result, map
    end if
end function

function GroupOffsets(left, right)
    for all offset $\in$ left.offsets do
        if (lseg $\leftarrow$ left[offset]) is Lseg then
            realOffsets $\leftarrow$ lseg.lambda[lseg.src].offsets
            yield offset, (left[offset], right[realOffsets])
        else if (lseg $\leftarrow$ right[offset]) is Lseg then
            realOffsets $\leftarrow$ lseg.lambda[lseg.src].offsets
            yield offset, (left[realOffsets], right[offset])
        else
            yield offset, ({left[offset]}, {right[offset]})
        end if
    end for
end function
```

lambda source variable `$src` and one from `$src+8`, the variable `$x` has pointers at offset 8 (8+0) and 16 (8+8), even if this might not be immediately evident from the keys of its type). If a mismatch is detected and one of the sides is an empty type (ie. it does not have any offsets), abstraction is requested (assuming that this pointer is the end of a chain and further pointers are simply not known at this point). If neither type is empty but offsets do not match, the join fails.

If all of the above steps succeed, the join succeeds.

Mathematically, this process can be seen as grouping offsets into subsets according to their value on the side with the bigger group (where pointers to variables have a group size of one containing the corresponding offset, and pointers to list segments correspond to the set of offsets present in the source type of the list segment's lambda, shifted by the initial offset). These sets are then split into separate structures, respectively, and their values are subsequently joined independently of each other (either two variables as usual, or list segments with a matching pointer structure or another list segment). If an offset in a group only exists on one side, the join fails. This process is visualized in Figure 5.4.

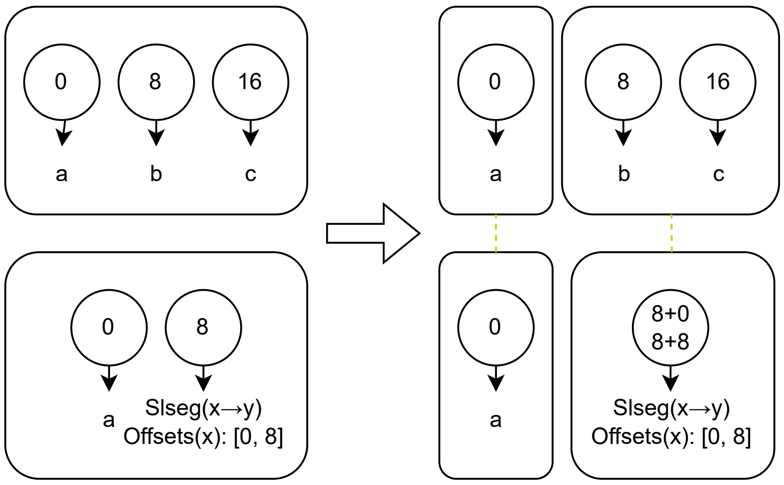


Figure 5.4: Joining two pointer types

For example, take the type map in Listing 5.4, joining `%l1` and `%l2`: First, offsets are split into groups $\{0\}$ and $\{8\}$. For each group, sizes of fields (trivial in this case: $8 \bowtie 8 = 8$) and targets are joined. `%l3` $\bowtie$ `%l5` results in `Scalar(?)`, while `%l4` $\bowtie$ `Slseg(...)` yields the same list segment (abstracting the single pointer into the segment). Now, all children are joined successfully and all offsets match (since the list segment does not define any additional offsets), and the join succeeds into the map seen in listing 5.5.

The join function rules for joining pointers are summarized in equations starting at 5.28.

Listing 5.4: Example type map with pointers

```
%l2: {
  0-(8)->%l5,
  8-(8)->Slseg(%l6, lambda(%l1, %l2) {
   %l1: {
     0-(8)->%l2
   }
   %l2: *
  })
}
%l3: Scalar(10)
%l4: {
  0-(8)->%l6
}
%l5: Scalar(11)
%l6: NULL
```

Listing 5.5: Example joined type map

```
%l1: {
  0-(8)->%l3,
  8-(8)->Slseg(%l6, lambda(%l1, %l2) {
   %l1: {
     0-(8)->%l2
   }
   %l2: *
  })
} (=%l2)
%l2: {...} (=%l1)
%l3: Scalar(?) (=%l5)
%l5: Scalar(?) (=%l3)
%l6: NULL
```

$$
\begin{aligned}
&\bowtie(tm, Ptr(p_1), Ptr(p_2)) && && (5.28)\\
&\quad = tm'', Ptr(\{(sof', tar')\}) && |p_1 = \{(sof_1, tar_1)\}, p_2 = \{(sof_2, tar_2)\}, && (5.29)\\
& && |tar_1, tar_2 \text{ are not list segments}, && (5.30)\\
& && | \bowtie (tm, sof_1, sof_2) = (tm', sof'), && (5.31)\\
& && | \bowtie (tm', tar_1, tar_2) = (tm'', tar') && (5.32)\\
&\quad = tm_n, Ptr(\forall (l, r) \in pg(p_1, p_2). && && (5.33)\\
&\qquad \bowtie (tm_{i-1}, l, r)) && |tm_0 = tm, n = |pg(p_1, p_2)|, && (5.34)\\
& && |tm_i = \text{type map result from ith join}, && (5.35)\\
& && |\text{no sub-results are TryAbstraction} && (5.36)\\
&\quad = \bowtie (tm, Ptr(p_1 \cup \{o \mapsto ls\}), Ptr(p_2)) && |\text{offset o requires abstraction (left side w.l.o.g.)}, && (5.37)\\
& && |ls = abstract(tm, o, p_1, p_2[o]) && (5.38)
\end{aligned}
$$

5.2.6 Joining list segments

When joining list segments, there are two base cases: A list segment is either joined with another list segment or with an unabstracted pointer type. These two options are described in this section, starting with the latter. After explaining how to join singly-linked segments, a short overview of how doubly-linked segments could be joined is given. The function for joining singly-linked segments can be seen in Algorithm 5.

This last part of the join is formally defined in Equation 5.39 and ones following it:

$$
\begin{aligned}
&\bowtie(tm, Slseg(a, \lambda_1), Slseg(b, \lambda_2)) && && (5.39)\\
&\quad = tm', Slseg(z, \lambda') && | \bowtie (\lambda_1 \cup \lambda_2, \lambda_1^{src}, \lambda_2^{src}) = (\lambda', src'), && (5.40)\\
& && | \bowtie (tm, a, Slseg(b, \lambda')) = (tm', z) && (5.41)\\
& && |(\text{swap a/b if Incompatible}) && (5.42)\\
& && | \bowtie (tm, a, b) = (tm', z) \text{ if both Incompatible} && (5.43)\\
&\bowtie(tm, Slseg(a, \lambda), t_2 = Ptr(p)) && && (5.44)\\
&\quad = \bowtie (tm' \backslash \lambda, Slseg(a, \lambda), t') && | \bowtie (tm \cup \lambda, \lambda^{src}, t_2) = (tm', t), && (5.45)\\
& && |tm' \models t' = \lambda^{dst} && (5.46)\\
&\quad = tm', Slseg(a, \lambda) && |\text{join above Incompatible}, && (5.47)\\
& && | \bowtie (tm, a, t_2) = (tm', t), && (5.48)\\
& && |\text{list segment previously joined with pointer to } t_2 && (5.49)
\end{aligned}
$$

Algorithm 5 Function for joining list segments

```
function JoinLseg(left, right, map)
    if left is Ptr then return JoinLseg(right, left, map)
    end if
    if right is Ptr then
        inlined $\leftarrow$ map $\cup$ freshVars(left.lambda)
        stepResult, stepJoined $\leftarrow$ Join(left.src, right, inlined)
        if stepResult $\in$ {Incompatible, TryAbstraction} then return Incompatible
        end if
        nextLink $\leftarrow$ var from map joined with left.dest
        updatedMap $\leftarrow$ stepJoined $\setminus$ left.lambda
        prunedMap $\leftarrow$ FilterReachableFromPVars(updatedMap $\setminus$ right)
        if prunedMap has pointers to right or right points to a PVar nextLink does
           not then                                                ▷ split segment
            if first join for this lseg then return Incompatible ▷ no empty segments
            end if
            splitJoinResult, splitJoinMap $\leftarrow$ Join(left, nextLink, map)
            if splitJoinResult $\in$ {Incompatible, TryAbstraction} then
                return Join(left.target, right, map)
            end if
            return (left with dest := right), splitJoinMap
        else if nextLink already encountered for this segment then       ▷ cycle
            return Incompatible
        else if nextLink = left.target then return left, prunedMap
        else return Join(left, nextLink, prunedMap)
        end if
    else
        combined $\leftarrow$ left.lambda $\cup$ freshVars(right.lambda)
        sourceRes, newLambda $\leftarrow$ Join(left.src, right.src, combined)
        targetRes, newLambda $\leftarrow$ Join(left.dest, right.dest, newLambda)
        if {sourceRes, targetRes} $\cap$ {Incompatible, TryAbstraction} $\neq \emptyset$ then
            return Incompatible
        end if
        outLseg $\leftarrow$ right with lambda := newLambda
        leftLongerRes, leftLongerMap $\leftarrow$ Join(left.target, outLseg, map)
        if leftLongerRes $\in$ {Incompatible, TryAbstraction} then
            outLseg $\leftarrow$ left with lambda := newLambda           ▷ other target
            rightLongerRes, rightLongerMap $\leftarrow$ Join(right.target, outLseg, map)
            if rightLongerRes $\in$ {Incompatible, TryAbstraction} then
                return Join(left.target, right.target, map)
            else return outLseg, rightLongerMap
            end if
        else return outLseg, leftLongerMap
        end if
    end if
end function
```

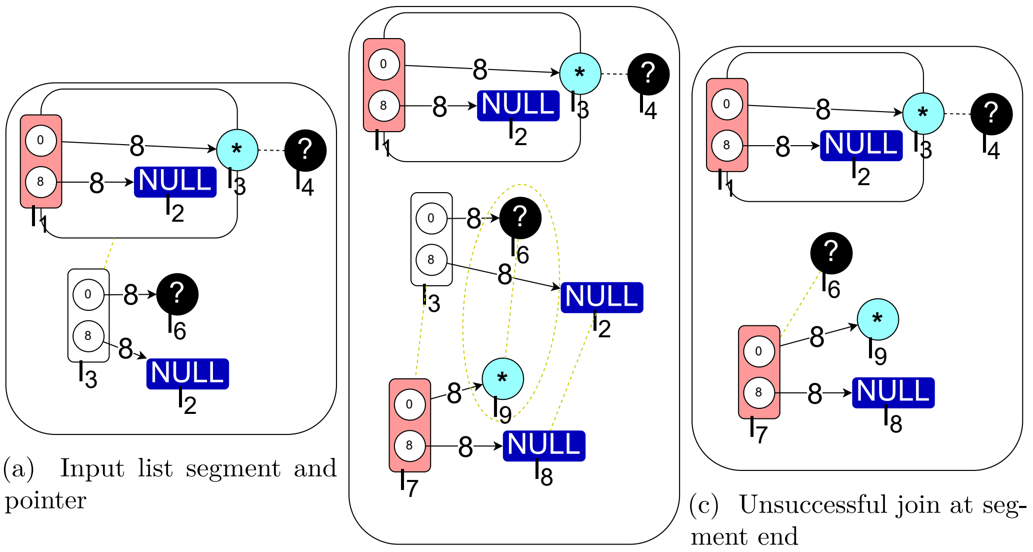
If a singly linked list segment is to be joined with a pointer type, the main challenge is to verify whether the lambda of the list segment is compatible with the said type and to find the next link starting from the pointer when this lambda is used. This is accomplished by taking the lambda, giving all available variables in it fresh names that do not yet occur in its parent type map and inlining it in the parent type map (while replacing outer types with their variable type equivalent), calling the join algorithm with the source variable of the lambda and the pointer type to join. Then, in case this succeeds, the next link can simply be found by looking at what the lambda destination variable has been joined with. Four possibilities can occur during this step:

- The join succeeds without a problem, and the link is accepted.
- The join succeeds, but there are pointers into the segment at this point or program variables would be lost by collapsing this pointer into the segment; in this case, the segment needs to be split into two.
- When the same situation as in the second option arises, but splitting is not possible (e.g. because the segment comes from a recent abstraction from a single pointer and thus cannot possibly be longer than one step), the only thing left to try is to assume that the current variable is already the end of the segment and to join it with the target variable of the list segment. This is only possible after at least one iteration of joining has been successfully performed and one or more pointers have been collapsed into the segment.
- The join fails, e.g. because there are additional pointers from the given variable.

If the next link is found, the pointer is collapsed into the segment and the algorithm repeats from the said variable. However, there are now some additional checks to be done:

- If the variable has already been encountered in the current list segment merge process, there is a cycle, and the join must fail (since target variables are joined only after growing a segment fails, and there is no cyclic list predicate without a target).
- If the current variable is equivalent to the list segment target variable, the end has been reached and the algorithm can terminate.

An example of joining a singly-linked segment with a pointer can be seen in Figure 5.5. 5.5a shows the original state, after which the list segment is inlined into its parent map with new variable names l_7, l_8, and l_9 (5.5b). Here, the yellow connections indicate joins and the yellow circle shows the newly discovered next link (l_6). After this, the process is repeated from the said variable, at which the join fails and l_6 is accepted as the segment end, ie. trivially joined with l_4 (5.5c).



(a) Input list segment and pointer

(b) Unwrapped list segment

(c) Unsuccessful join at segment end

Figure 5.5: Joining list segment with pointer type

In case the segment has to be split, a second list segment is assumed to start at the current variable, and joining continues from there. If the join fails altogether, the algorithm tries to join the list segment target variable with the current variable (if allowed) and fails otherwise.

The other case to consider is two list segments being joined. Here, first, their lambdas have to be compatible, which is checked by a similar algorithm as before (combining them into one type map and joining source variables), explicitly joining target variables (which are known) this time as well. Then there are the three cases in which either one segment is longer than the other, or they are the same length. Therefore, the algorithm tries to join one of the segments (the one assumed to be the longer one) with the target variable of the other (the one assumed to be the shorter one) to try and find the missing part that assimilates the lengths of the two. If this fails, the other way around is tried. If neither is possible, the two target variables are joined (assuming the segments are the same length), or the join fails.

Doubly-linked segments can be joined in a relatively similar way. To join with a pointer, the segment lambda is inserted into its parent type map, and the `current` variable is joined with the pointer type in question to find not only one but possibly two successor links. In each direction, joining is repeated until impossible, and target variables are joined. Additionally, backlinks have to be verified for each additional link to be collapsed into the segment.

Joining two doubly-linked segments is equally similar to the singly-linked case, again with the modification that joining must commence in two directions instead of only one.

The remaining question is how to join one singly- with one doubly-linked segment, or if this should be done at all. The problem here is that naturally, singly linked segments lack information about a backlink. In practice, however, this could simply mean that such a link has not been discovered before abstraction has been applied; during join, there is however no way of knowing if this was the case, or if the link really is not there for a certain state. Therefore, joining two such states should yield an incompatible result.

5.2.7 Joining constraints

Joining constraints follows a simple set of rules:

- The join cannot introduce an equality between program variables (directly or via aliases) that was not previously there.
- Null, not-null, constant value and inequality constraints present only on one side are discarded, except for the case where both sides have a distinct constant value (in which case the join fails).
- All other constraints are directly inherited.

The pseudocode for this procedure can be seen in Algorithm 6.

Algorithm 6 Function for joining constraints

```
1: function JoinConstraints(left, right)
2:     if (left $\cup$ right) introduces alias between PVars (transitively) then return $\bot$
3:     else if Const(x) $\in$ left and Const(y) $\in$ right and x $\neq$ y then return $\bot$
4:     else
5:         filtered $\leftarrow$ {Null, NotNull, Const(*), NotEq(*)}
6:         return ((left $\cup$ right) $\setminus$ filtered) $\cup$ (left $\cap$ right)
7:     end if
8: end function
```

This, as for some cases in Section 5.2.4, is not necessarily correct on both sides of a state, but has proven to be effective in real-life scenarios - see Section 5.3.

5.2.8 Abstraction

Abstraction is tried in cases where one side contains pointers and one side does not, assuming that this indicates pointers in a list segment that have not been traversed and are thus not yet known. This can happen in the case where the unknown side has the unknown type, or where it has an empty pointer type (ie. it is known that this is a pointer, but not what it points to). In such cases, a lambda is synthesized as described in this section and the join continues with the corresponding list segment.

To synthesize a new lambda, the original type map is taken and the current variable is marked as the source variable. The type of the destination variable is replaced with the wildcard type (which can be joined with anything). Then, all variables not reachable from the source variable (that are, therefore, not relevant for the lambda), as well as aliases (not necessary inside of a lambda because there can be no program variables), are dropped. In addition, all further occurrences of the start variable are replaced with an outer variable pointing to the list start under the assumption that pointers back to the list start are common (and cycles inside of a lambda cannot be handled properly anyway). This is described in Algorithm 7.

Algorithm 7 Function for abstraction

```
1:  function ABSTRACT(type, map, offset)
2:      result ← Slseg
3:      result.lambda ← map
4:      result.lambda[type[offset]] ← *
5:      result.lambda ← REACHABLEFROM(result.lambda, type)
6:      result.lambda ← REMOVEALIASES(result.lambda)
7:      result.lambda ← result.lambda{type ↦ Outer(type)}
8:      result.src ← type
9:      result.dest ← type[offset]
10:     result.target ← type[offset]
11:     return result
12: end function
```

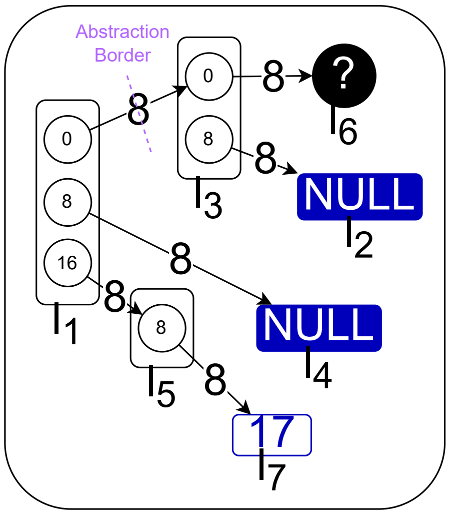
This is, of course, an abstraction that makes a set of assumptions that might not be true in every case. Most prominently, all abstractions produce singly linked segments, and lambdas that step over more than one pointer (e.g. for loops that go two steps for each iteration) will not be found. However, wrong results will be discarded during validation as described in section 5.3, causing the join to fail (since the right assumptions could not be made anyway).

This newly created list segment is added to the host type map in stead of the first pointer of the shorter segment, and the algorithm for joining list segments with pointers is applied.

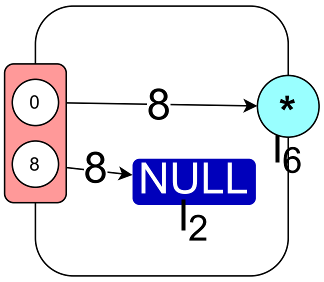
Figure 5.6 shows an example of how a list segment is generated. Abstraction is requested for l_3 in 5.6a, so all variables not reachable from l_3 (l_1, l_4, l_5, l_7) are dropped. l_3 is selected as the source variable (being the abstraction source), and the pointer target at the respective offset (where abstraction has been requested from), l_6, as the segment target. This results in the segment seen in 5.6b.

Formally, the abstraction function can be seen in Equation 5.50.

$$abstract(tm, o, p_1, p_2) = Slseg(p_1[o], reachable(tm, p_1[offsets(p_2)])\{p_1 \rightarrow p_1[o]\}) \tag{5.50}$$



(a) Input type map



(b) Output list segment

Figure 5.6: Example abstraction

5.2.9 Example

For example, take the two input type maps from Figure 5.7, representing a single step in a loop iterating through a linked list ($x \to l_3(\to l_4) \to l_7$). The pairs connected via yellow lines are attempted to be joined, starting at x and propagated through pointers at matching offsets. For l_1, l_2, and l_5, this is straightforward, but l_3^l and l_7^r cannot be directly merged, so abstraction is requested.

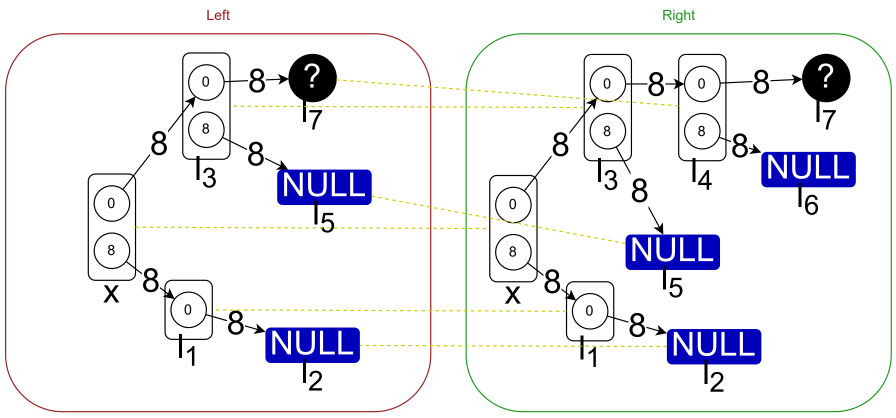


Figure 5.7: Example join input

The problematic pointer on the left (shorter) side is replaced with the corresponding list segment, as can be seen in Figure 5.8. Starting from its origin variable l_1, the lambda of the list segment is joined with the corresponding pointer on the opposite side to determine the next link, with l_4 as a result. This is repeated to get the link to l_7. Another try fails (since there is no link from l_7), and so targets (l_7^l and l_7^r) are joined. This trivially succeeds, so the pointers $l_3^r \to l_4 \to l_7$ are collapsed into the generated list segment. l_5 is now not referenced anymore, so the variable is removed. Since all of x's children are joined, joining x is also done, and the type map is finally joined into figure 5.9.

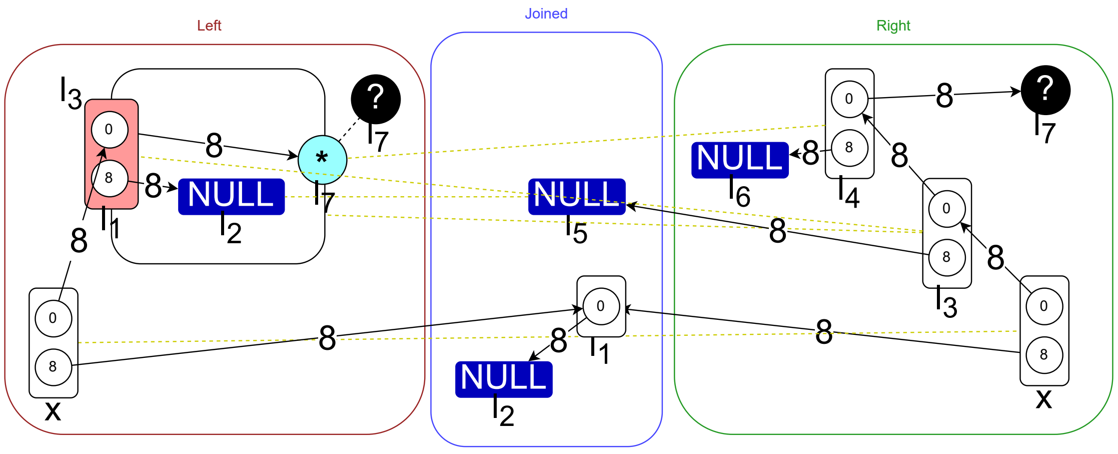


Figure 5.8: Example join after abstraction

Note that the list segment could be extended one link further in the beginning, in which case the second pointer (at offset 8) would have to be abstracted to a nested list segment ($x \to l_1 \to l_2$ / $l_1 \to l_2$); since this is not required for the join to succeed, however, the pointers are left as-is.

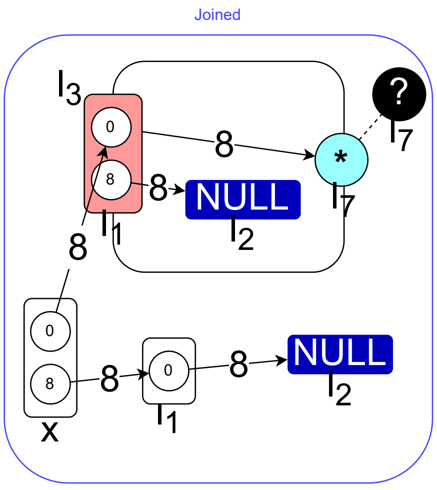


Figure 5.9: Example join result

5.2.10 Converting back

When a joined type map is available, it has to be converted back to a Broom formula; this is done for the missing and the current part separately.

First, points-to- and list-segment predicates are generated for all variables that have a pointer type (remember that list segments are always children of pointer types). If there is a "more important" variable (where named program variables are more important than special logical variables, which in turn are more important than logical variables; in each of these three groups, variables with a lower number are more important), no predicates will be generated (since an expression indicating equivalence to this variable suffices). Then, first, all child list segments are converted to list segment predicates, using the algorithm recursively for the lambda. Furthermore, outer types are replaced with fresh variables, which are inserted as parameters for the lambda. All offsets that are duplicated between the source variable in the lambda and the parent pointer type are discarded, except for where the parent points directly to a program variable. In the latter case, it is assumed that the last link of the list segment points to this program variable (ie. it is a pointer updated to the current position in every loop iteration), and the segment is unrolled once (this is verified later, see Section 5.3). All other members of the pointer type are converted directly to points-to-predicates in a straightforward way. Since the join at the time of writing will never emit complex types as pointer targets (but only scalar or variable types which may hide complex types), recursion is not necessary at this point.

Then, constraints are converted to expressions after being deduplicated between equivalent variables just like for spatial predicates. This is a very straightforward process that simply maps the different constraints to expressions (e.g. a constant or an alias constraint will become an equivalence, and a not-null constraint will become an inequivalence). In this step, the necessary equivalences for previously ignored "less important" variables are introduced. Here, also equivalences between program variables and logical variables only present on the other side (missing or current) are re-introduced from the original type map so as not to lose any connection between related variables from the pre- and the postcondition. For example, a list segment might have a logical variable as its target in the missing part, while in the current part, this variable is concretized in the form of a program variable, including an equivalence between the two variables. Simply emitting a list segment to the program variable, even though correct when only considering one side, drops the connection to the precondition, so the equivalence has to be included in the final output.

Lastly, types not handled previously (scalar, variable, and wildcard types) are converted to expressions. As for constraints, this is a simple mapping process.

All spatial predicates and expressions generated are then combined into a formula and returned.

5.3 Validating output

While most operations during the join process are correct on their own (and their result could simply be used straight away), some of the transformations take a few more liberties for the sake of speed and simplicity (which will in most cases be correct, but not necessarily). In addition, there might be some expressions the join simply cannot understand, which will then be copied to the output formula and verified only in the end (expression constraints).

To verify a join result, first, both the missing and current formula are checked for satisfiability using the solver, since an unsatisfiable formula can in no case be a sensible join result. If these checks succeed, the join is checked for correctness by checking for entailment between the two old and the new formulas. Only if this succeeds will the join return a result; otherwise, it will simply return Incompatible.

5.4 Entailment

Even after a successful, validated join, this does not mean that a fixed point for a loop has been found (since information is possibly lost during the join). Furthermore, some course of action has to be taken for states that could not be joined. Therefore, employing entailment somewhere in the process seems like a very natural move.

Experiments have shown that the best results can be reached by using one run of entailment before any joining is done to remove all potential fixed points, and another run on join results for further pruning. If the entailment before the join is not done, in some cases, cycles of joined states can occur that yield the exact same successor states after every loop iteration (and therefore the same join inputs and results) which cannot be pruned after the join, so the system never detects that a fixed point has been reached.

CHAPTER 6

Implementation

Figure 6.1 contains an overview of the most important parts of the join algorithm in the form of a call graph of the `Biabd.Join` OCaml module. Joining starts at the top (`join_states`) and continues along the edges, depending on the structure of the input data.

First, `get_all_types` constructs a type map by retrieving types for all available variables. This type map is passed to `merge_all_types`, which resolves all program variables and anchors using `merge_get_type_from_resolved`. This function is initially called with a variable type for the corresponding variable, but later also serves as an abstraction layer to be able to simply retrieve a type and guarantee that it is complete. This function, as well as most below it, returns a `type result` as introduced in Section 5.2.2 (`Incompatible`, `Type`, `TryAbstraction`, or `Compatible` for all functions other than the resolve function).

`Biabd.Join.types_to_formula` takes a merged type map and its original (unmerged) counterpart (to make sure no variables are incorrectly reused) and returns a formula to be used in the resulting state, once for the missing and once for the current part.

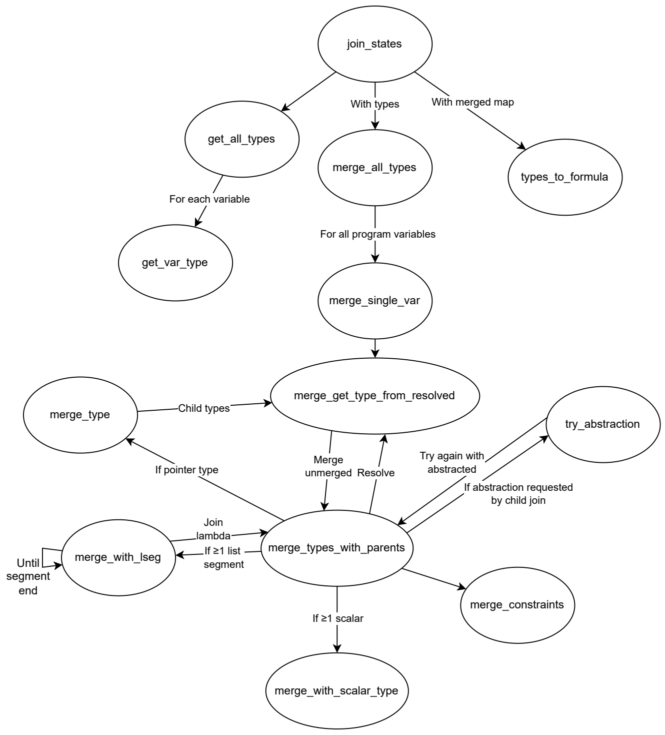


Figure 6.1: Overview of the algorithm in the form of a call graph

CHAPTER 7

Considered Variations

During development, a few variations in how joining is done have been considered and evaluated by comparing analysis results. For all of them, one option proved to clearly outdo the others, with the alternatives providing strictly worse or even incorrect results. This chapter introduces the substantial alterations considered and gives reasons as to why the current approach was chosen for the respective aspect.

7.1 Joining pointers with null

A main discussion point is how variables should be handled that have a valid pointer type on one side (possibly even with a not-null constraint) and that are null on the other. Other frameworks, like Predator[HKP+16], have explicit nullable types for these cases (called 0/1 abstract objects in Predator); this is not supported by Broom. Therefore, the base options here are to either lose the information that a variable can be null (simply dropping the null constraint) or to disallow joining in this case. A third option, which has been implemented and has yielded the best results, is to only allow such a join if the pointer type is empty, ie. nothing about its targets is known. This way, no one-sided pointers end up in the join result (offsets only present on one input side), but joining does not automatically fail simply because some variable is null (often because it is uninitialized).

7.2 Joining list segments with null

Similar to pointers, one could argue that joining list segments with null should not be possible in order not to create any empty segments. Furthermore, the list of offsets of a list segment can never be empty, since there has to be a link to the next element. Indeed, Broom does not allow empty segments, so joining one with null should not be possible. Joining a list segment with a pointer to null, however, is a different story: Here, a link is present (the pointer itself), and so if the rest of the lambda matches the source pointer type, a join is possible. This also leads to the next question, which is whether to try abstraction when a non-empty pointer is joined with null (making it possible to join null with the newly created list segment) or if the algorithm should simply fail in such a case.

7.3 Abstracting on pointer + null

As mentioned in section 7.2, a possibility is to request abstraction from the parent structure if a pointer type is to be joined with null, and to collapse both (the single-step segment to null as well as the multi-step segment containing the non-empty pointer type) into the subsequently created list segment. In theory, this sounds like a convincing idea, and results are certainly no less correct than when abstracting at any other arbitrary point (like what is done when the join algorithm is not used). However, enabling this interestingly causes results to degrade greatly because of wrong abstractions, which is not a problem if the same process is only allowed for unknown instead of null types. This is likely because many variables are initialized with null and held in some parent structure, but are in reality not part of any list segments. Unknown types, on the other hand, mostly occur when a pointer has not yet been discovered, usually as part of a loop (which usually iterates over some kind of list-like structure).

7.4 Shared expressions in abstraction

List segments allow for shared expressions, ie. expressions that are globally shared between all links in a segment. During abstraction, a choice has to be made as to which variables to include directly in the lambda and which ones to reference via shared expressions (or, during join, outer variables). Options here are to inline all variables, share a certain class of variables (such as all pointers not directly related to getting to the next list entry), or share variables related to the segment in a certain way. In tests, it became apparent that pointers back to the list head can be relatively common; therefore, references to the source variable are converted to outer variables during abstraction while all other types are added directly to the lambda.

7.5 Only allowing correct joins

Lastly, a big question during the whole project was how to get join results to be correct. The two relevant alternatives here are to either check the result with a solver (as is done in the final version) or to make all operations during join provably correct. The latter, however, would mean that either a lot of information is lost (with the output state containing the intersection of information from the input), or most if not all joins are not possible (failing joins as soon as too much information is lost). Therefore, accepting the overhead of a solver check has been chosen as the superior option.

CHAPTER 8

Evaluation

The implementation of the join operator for Broom has been evaluated to grade the result quality both during implementation (to support choices made in the design of the algorithm, as explained in Chapter 7) and for the final result. This section presents evaluation results for the final join algorithm compared to the base case - Broom without a join operator, as introduced in Chapter 3. First, an overview of how comparisons were done is given in Section 8.1; the subsequent sections show comparisons between the two implementations categorized by different features. Finally, in Section 8.5, inputs that produce notably different results between the two approaches are listed and potential causes for the differences are explained.

The metrics used for comparison are the total runtime and memory usage of the Broom process per file analyzed, the number of join points encountered per file or function, as well as the quality of results, where a case in which a correct contract is found is better than one without it, just like for cases where a more specific postcondition or a less specific precondition in an otherwise equivalent contract is synthesized. Semantically equivalent contracts are equally good. For all other cases, contracts are considered incomparable; however, this situation has not occurred during evaluation.

8.1 Benchmark suite

To be able to evaluate the algorithm sensibly, a good input data set is required. There are many benchmark suites for various purposes readily available online; however, a choice was made here to use a very specific set of input programs rather than a generic set. Analysis done by Broom and thus cases relevant for the join operator are rather specific to linked lists; in generic data sets, a majority of cases would not yield a result since Broom is not built to handle the patterns encountered there. Instead, the test set used is one used for other tests with Broom before and is included in the repository[HPR$^+$19] under the `tests` folder. Here, there are functions specifically crafted for certain cases (which enables checking for specific improvements - for example, a function that traverses a linked list or a function that creates one, but also more complex examples), alongside code from real-world projects that rely heavily on linked lists, including part of the Linux kernel or several allocators (that keep track of allocated blocks in linked-list-like structures), like `tinyalloc` or the FreeBSD implementation for `malloc`.

Altogether, this adds up to 119 C files, with 572 functions in 7038 lines of code (excluding blank lines and comments, as well as files with no join points).

8.2 Number of join points

The first relevant statistic considered is whether the join operator can decrease the number of loop iterations needed to arrive at a stable state, ie. how many times the analysis arrives at a join point. Interestingly, this number varies relatively little, with only 12 of the examples indicating any difference at all - seven of them needing more and five of them less join points. In the cases where fewer joins are needed, the same contracts are synthesized in all cases. For the examples where more joins are needed, some produce the same results (3), but in a few cases, the output now contains contracts not previously found, indicating that the additional joins are in reality a positive thing needed for better results.

8.3 Runtime

The second feature to compare between the two variants is the runtime needed to calculate final contracts. All benchmarks here were done on a computer with an AMD Ryzen 9 3900X processor and 64 Gigabytes of RAM running openSUSE Leap 15.4 (kernel 5.14.21 / 64 bit) in a Docker container based on ubuntu:22.04 (from the Dockerfile included in the Broom repository) with OCaml version 4.09.1. Measurements were taken using `time -v`, executing Broom with and without the `--join` flag and default parameters otherwise (most notably an entailment limit, ie. the limit of how many runs through a loop are done, of five).

Out of the 119 input files, 7 (6%) completed noticeably faster when run with the join algorithm than without, 5 (4.2%) of them with a speedup of more than 10%. However, 42 (35%) of the files took more than 10% longer than before. This is to be expected since a failed join takes quite a lot of time without any contribution to the analysis process. A graph summarizing the time comparisons per input file can be found in Figure 8.1. This shows that while the execution time is in general higher, there are some cases where an improvement can be observed.

As for memory usage, results are similar but differences are in most cases less drastic than for execution time. Here, only six (5%) input files used more than 10% less memory, and only 14 (13%) files needed more than 10% more. A comparison for including the

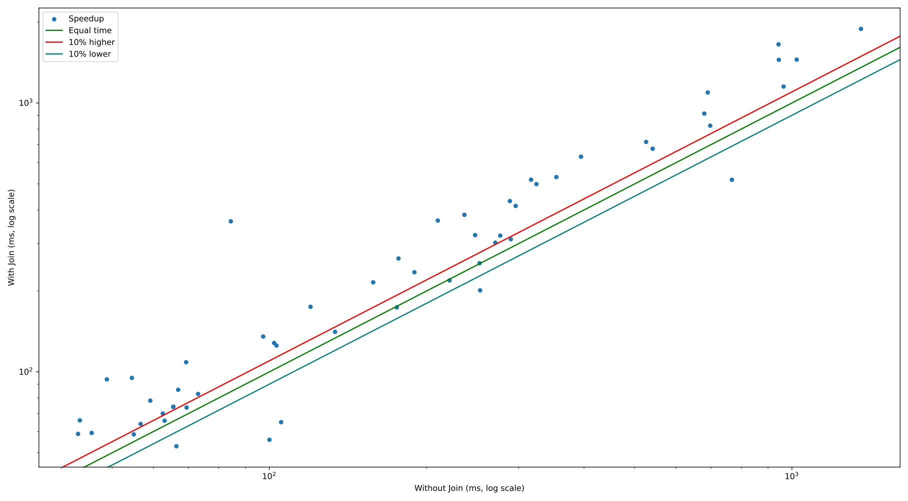


Figure 8.1: Execution time with and without the join algorithm

different input files can be found in Figure 8.2. All numbers are for the entire Broom process including calls to the solver.

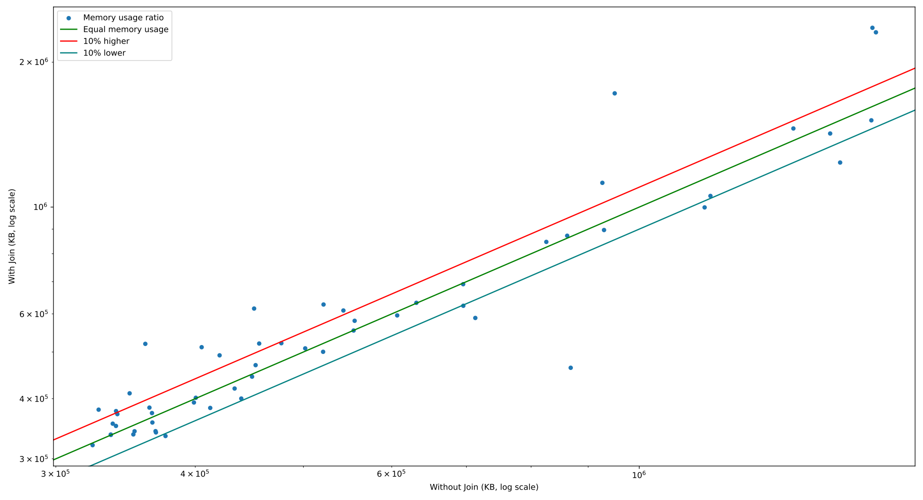


Figure 8.2: Maximum memory usage with and without the join algorithm

All examples where a significant difference in either of the two metrics was observed can be seen in Table 8.1, together with concrete values for these features.

Table 8.1: Time and memory consumption comparison

File	Time (sec)			Memory (KB)		
	$\bowtie$	$\not\bowtie$	Ratio	$\bowtie$	$\not\bowtie$	Ratio
sll-shared-sll-before.c	55.90	100.04	0.56	320264	323640	0.99
sll-shared-sll-after.c	65.01	105.31	0.62	334564	376188	0.89
intrusive-list-top.c	518.07	767.69	0.67	463676	868104	0.53
linux-list.c	200.97	253.09	0.79	500508	520784	0.96
test-0233.c	52.90	66.38	0.80	400220	439868	0.91
test-0192.c	58.56	55.02	1.06	492136	420608	1.17
intrusive-list-minimal-example2.c	69.98	62.52	1.12	354908	337372	1.05
intrusive.c	64.00	56.72	1.13	379504	327692	1.16
sll.c	74.01	65.44	1.13	351076	339628	1.03
sll-main__gcc.c	82.76	73.05	1.13	376920	339688	1.11
sll-main.c	74.25	65.51	1.13	371500	340604	1.09
dll-fst-data.c	321.22	276.56	1.16	691592	695324	0.99
linux-hlist-fst-data.c	823.22	697.61	1.18	846600	825500	1.03
linux-hlist-lst-data.c	1151.04	963.87	1.19	1237484	1513716	0.82
predator-test-0156-no-include.c	125.27	103.20	1.21	521536	477788	1.09
sll-shared-after.c	234.70	189.44	1.24	382664	412564	0.93
dll-middle-data.c	675.83	541.38	1.25	998520	1144448	0.87
predator-test-0156.c	128.05	102.11	1.25	469568	453064	1.04
call-01-ok__gcc.c	85.78	66.90	1.28	511672	405320	1.26
dll-trans__gcc.c	59.29	45.70	1.30	342452	368528	0.93
dll-lst-data.c	322.52	247.61	1.30	632632	631308	1.00
dll.c	78.17	59.16	1.32	342272	352908	0.97
linux-list-lst-data.c	913.20	679.64	1.34	1514428	1614460	0.94
linux-list-fst-data.c	716.41	525.93	1.36	1123048	926724	1.21
sll-nested-sll-inline.c	215.23	158.00	1.36	444408	449752	0.99
dll-trans.c	58.80	43.04	1.37	340368	368836	0.92
tinyalloc–orig.c	2666.60	1949.66	1.37	1114072	1168068	0.95
sll-shared-before.c	135.33	97.31	1.39	373380	365836	1.02
sll-nested-sll-lst-data.c	1886.19	1355.43	1.39	1722976	950624	1.81
new__malloc–orig.c	414.13	295.97	1.40	2357280	1618036	1.46
linux-hlist-middle-data.c	1449.46	1021.32	1.42	1421792	1482892	0.96
dll-backwards.c	174.59	119.88	1.46	401712	400468	1.00
sll-lst-data.c	264.06	176.60	1.50	520684	456416	1.14
new__malloc–simplified.c	431.73	288.54	1.50	1456412	1374516	1.06
dll-lst-shared.c	530.22	354.14	1.50	595672	606824	0.98
tinyalloc–simplified.c	2820.79	1858.92	1.52	1181744	1125384	1.05
sll.c	66.02	43.38	1.52	337136	352028	0.96
linux-list-middle-data.c	1447.41	943.87	1.53	2306968	1629796	1.42
sll-lst-shared.c	499.21	324.37	1.54	588448	712976	0.83

Table 8.1: Time and memory consumption comparison

File	Time (sec)			Memory (KB)		
	$\bowtie$	$\not\bowtie$	Ratio	$\bowtie$	$\not\bowtie$	Ratio
linux-list-t1+traverse.c	108.61	69.29	1.57	392764	398916	0.98
sll-middle-data.c	631.27	394.78	1.60	871932	861844	1.01
sll-fst-shared.c	383.68	236.15	1.62	580504	555804	1.04
dll-fst-shared.c	518.36	316.81	1.64	623764	695536	0.90
sll-fst-data.c	365.38	210.08	1.74	609824	543048	1.12
linux-list-traverse.c	94.94	54.56	1.74	383256	363904	1.05
sll-middle-shared.c	1652.06	942.61	1.75	896108	929916	0.96
sll-headptr.c	93.77	48.86	1.92	410364	349332	1.17
sll-shared-sll.c	362.95	84.38	4.30	519888	360836	1.44

The most probable explanation for longer execution times is that a lot of joins are performed, most or all of which are unsuccessful. If, in addition, failure is detected late in the algorithm (after a lot of variables have already been joined), a lot of time is used without any gain.

Examples that are faster with the join algorithm can be explained with the fact that joined formulas are both more concise (filtering out unnecessary clauses), possibly making it easier for the solver to work with them, and that there are fewer formulas to begin with, resulting in the symbolic execution procedure to be called fewer times. In addition, in two of the cases loops terminate in fewer iterations, eliminating the need to run all parts of the system (symbolic execution, bi-abduction,. . .) again.

8.4 Result quality

With the previous evaluation criteria being how fast a result is reached (in time or the number of operations), the other important aspect is the quality of results, ie. which contracts are found and which ones are not. This section gives a quantitative overview, while the next one talks about the most important differences.

As expected, the contracts generated with and without the join algorithm are the same for most functions; in total, there are 22 functions with differing contracts. Two of these functions have contracts that are semantically the same but differ syntactically, for two functions contracts are found with an entailment limit of five where this is not possible without the join, and for the others, additional contracts are returned (28 contracts in total).

It is also worth noting that with an older version of Broom, contracts used to be lost in certain cases; this is discussed in Section 8.5.1.

8.5 Notable differences

This section explains some of the most interesting differences encountered in the analysis output and tries to give reasons as to how they came to be. In addition to the contracts given below, contracts for calling functions (e.g. a `main` function calling a list creation function) could in some cases be found, now that all called functions had valid contracts; these cases are trivial and thus not discussed in further detail.

8.5.1 Nested lists

The most problematic case during the evaluation of the join algorithm arose when analyzing functions traversing nested lists: For the second level (counting from the innermost list), additional slightly wrong (unsound) contracts were generated (with an additional step in the list, expressed using points-to-predicates, only on one contract side), which leads to the system not being able to generate any contracts at all for all further nesting levels.

The explanation for this is the fact that variable scoping was not implemented in Broom at the time of analysis (or rather only on a function level); this leads to the join being forced to unwrap the last list segment (since it has a pointer to the current inner list, which is, however, not even needed anymore and could simply be dropped), which results in an explicit additional segment during the second run (ie. only in the postcondition). Indeed, after a live variable analysis was added to Broom and dead variables were removed from states (ie. converted to logical variables), this problem was solved.

8.5.2 List destruction

One of the main improvements the join algorithm was able to achieve was during the analysis of list deallocation functions. For example, for the `sll_clear` function in Listing 8.1, previously no contracts could be found; with the join algorithm, however, the two contracts in Listing 8.2, one for an empty list and one for non-empty ones, are generated.

This is because two successor states in the list are joined into a single state with a list segment where entailment itself repeatedly does not prune the new state, presumably because of an additional pure predicate for the new link that is simply filtered out during the join (where information is lost, but the state is still verified to be correct). Then, after this join succeeds, this entailment does as well (on the join result instead of the initially generated state), marking the state as stable and leading to the termination of the loop.

Listing 8.1: sll_clear function

```
struct sll_item {
    struct sll_item *next;
    int data;
};
void sll_clear(struct sll_item **plist)
{
    struct sll_item *p = (*plist);
    while (p != NULL) {
        struct sll_item *next = p->next;
        free(p);
        p = next;
    }
    (*plist) = NULL;
 }
```

Listing 8.2: Contracts for sll_clear (shortened for brevity)

```
Count of Contract EVARS:5
LHS:%c1-(8)->%c2 & (%mF14:plist_anch=%c1) & (%c2=NULL) & ...
RHS:%c1-(8)->%c5 & (%mF14:plist_anch=%c1) & (%c5=NULL) & ...
Prog. VARS moves:
Count of Contract EVARS:14
LHS:%c1-(8)->%c2 *
Slseg(%c2,%c11,lambda[%l6,%l10]:{
      %l6-(8)->%l10 * %l1-(%l2-8)->Undef &
      (%l6!=NULL) & (base(%l6)=%l6) & (len(%l6)=%l2) & (base(%l6)=base
         (%l1)) & (%l1=(%l6+8)) & ...
},[]) &
(%mF14:plist_anch=%c1) & (%c2!=NULL) & (base(%c2)=%c2) & (base(%c2)=
   base(%c5)) & (%c5=(%c2+8)) & (%c11=NULL) & ...
RHS:%c1-(8)->%c14 & (%mF14:plist_anch=%c1) & (base(%c2)=base(%c5)) & (
   %c5=(%c2+8)) & (%c2!=NULL) & (base(%c2)=%c2) & freed(%c2) & (%c11=
   NULL) & (%c14=NULL) & ...
Prog. VARS moves:
```

CHAPTER 9

Conclusion

While shape analysis can certainly work using only opportunistic abstraction and entailment via a solver at join points, in some cases having a join operator that unifies formulas based on their structure and applies abstraction intelligently can result in better outcomes. Results do, though, vary greatly depending on the exact parameters and rules used for joining, where in many cases the effects are far more negative than for others, so choices for such an algorithm have to be made carefully and evaluated on real-world examples. While join points have been analyzed extensively for other types of analyses, literature on joining during shape analysis is relatively scarce, with only a handful of approaches described and evaluated, each implemented as part of a certain analysis ecosystem.

The algorithm described in this thesis and implemented as part of the Broom project yields improved results, as confirmed with a test set of characteristic examples. The operator enhances the analysis process, and even in cases where no improvement is observed, the downsides are usually only minor and the results are mostly equivalent to the base case. Nevertheless, such an operator cannot work wonders and in many cases, no function contracts can be found still.

In addition to the algorithm used when joining, integration into the analysis system plays an important part in ensuring good outcomes. Steps such as abstraction or pruning formulas can occur before, after, or even during the join, with greatly varying levels of success. In addition, entailment can immensely aid and simplify the join process by proving correctness or finding redundant states.

CHAPTER 10

Future Work

Based on the work in this thesis, an important point to be implemented and evaluated is the joining of doubly-linked list segments; how such an algorithm might look has been introduced above. Here, an important point is to analyze the effects of such an implementation, just like what was done for singly-linked segments here.

In addition, some research might be done on how to join more than two states at once, if applicable. This is also closely intertwined with the question of which states to join, which was answered in a relatively short manner in this thesis. With some additional research, answers to these questions might bring additional improvements to the process.

List of Figures

List of Tables

List of Algorithms

List of Listings

Bibliography

[Atk10] Robert Atkey. Amortised resource analysis with separation logic. In *European Symposium on Programming*, pages 85–103. Springer, 2010.

[BBB$^+$22] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[BBC08] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. *ACM SIGPLAN Notices*, 43(1):101–112, 2008.

[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 *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*, pages 178–192. Springer, 2007.

[BIP10] Marius Bozga, Radu Iosif, and Swann Perarnau. Quantitative separation logic and programs with lists. *Journal of Automated Reasoning*, 45(2):131–156, 2010.

[CDOY06] Cristiano Calcagno, Dino Distefano, Peter W O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Static Analysis: 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006. Proceedings 13*, pages 182–203. Springer, 2006.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W O'hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM (JACM)*, 58(6):1–66, 2011.

[Chl11] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 234–245, 2011.

[CL20] Christopher Curry and Quang Loc Le. Bi-Abduction for Shapes with Ordered Data. *arXiv preprint arXiv:2006.10439*, 2020.

[CLQ19] Christopher Curry, Quang Loc Le, and Shengchao Qin. Bi-abductive inference for shape and ordering properties. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 220–225. IEEE, 2019.

[Cop14] Patrick T Copeland. Using State Merging and State Pruning to Address the Path Explosion Problem Faced by Symbolic Execution. 2014.

[CRN07] Bor-Yuh Evan Chang, Xavier Rival, and George C Necula. Shape analysis with structural invariant checkers. In *International Static Analysis Symposium*, pages 384–401. Springer, 2007.

[DDZ94] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software: Practice and Experience*, 24(6):527–542, 1994.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DMP$^+$13] Kamil Dudka, Petr Muller, Petr Peringer, Veronika Šoková, and Tomáš Vojnar. Algorithmic Details behind the Predator Shape Analyser Based on Symbolic Memory Graphs. 2013.

[DOY06] Dino Distefano, Peter W O'hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems: 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-April 2, 2006. Proceedings 12*, pages 287–302. Springer, 2006.

[DPV11] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 372–378. Springer, 2011.

[DPV13] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Byte-precise verification of low-level list manipulation. In *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings 20*, pages 215–237. Springer, 2013.

[DPV14] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A Shape Analyzer Based on Symbolic Memory Graphs: (Competition Contribution). In

International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 412–414. Springer, 2014.

[DVP$^+$16] Kamil Dudka, Tomáš Vojnar, Petr Peringer, Petr Müller, Veronika Šoková, and Michal Kotoun. Predator. `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/`, 2016. Accessed: 2024-02-10.

[HKP$^+$16] Lukáš Holík, Michal Kotoun, Petr Peringer, Veronika Šoková, Marek Trtík, and Tomáš Vojnar. Predator shape analysis tool suite. In *Hardware and Software: Verification and Testing: 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings 12*, pages 202–209. Springer, 2016.

[HPR$^+$19] Lukáš Holík, Petr Peringer, Adam Rogalewicz, Veronika Šoková, Tomáš Vojnar, and Florian Zuleger. A Static Analyzer for C Based on Separation Logic and the Principle of Bi-Abductive Reasoning. `https://pajda.fit.vutbr.cz/rogalew/broom`, 2019. Accessed: 2024-02-03.

[HPR$^+$22] Lukáš Holík, Petr Peringer, Adam Rogalewicz, Veronika Šoková, Tomáš Vojnar, and Florian Zuleger. Low-Level Bi-Abduction. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:30, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[HSS09] Trevor Hansen, Peter Schachte, and Harald Søndergaard. State joining and splitting for the symbolic execution of binaries. In *International Workshop on Runtime Verification*, pages 76–92. Springer, 2009.

[Kai23] David Kaindlstorfer. Enhancing Abstraction and Symbolic Execution for Shape Analysis of C-Programs operating on Linked Lists, 2023.

[KKBC12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012.

[KSV10] Jörg Kreiker, Helmut Seidl, and Vesal Vojdani. Shape analysis of low-level C with overlapping structures. In *Verification, Model Checking, and Abstract Interpretation: 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings 11*, pages 214–230. Springer, 2010.

[LBCR17] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. Semantic-directed clumping of disjunctive abstract states. *ACM SIGPLAN Notices*, 52(1):32–45, 2017.

[LC02] Woo Hyong Lee and Morris Chang. A study of dynamic memory management in C++ programs. *Computer Languages, Systems & Structures*, 28(3):237–272, 2002.

[LGQC14] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 52–68. Springer, 2014.

[Min06] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 54–63, 2006.

[MSRF04] Roman Manevich, Mooly Sagiv, Ganesan Ramalingam, and John Field. Partially disjunctive heap abstraction. In *Static Analysis: 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004. Proceedings 11*, pages 265–279. Springer, 2004.

[O'H19] Peter O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[OP99] Peter W. O'Hearn and David J. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[PZ22] Jens Pagel and Florian Zuleger. Strong-separation logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(3):1–40, 2022.

[Rey02] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[SHB16] Dominic Scheurer, Reiner Hähnle, and Richard Bubel. A general lattice model for merging symbolic execution branches. In *Formal Methods and Software Engineering: 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14-18, 2016, Proceedings 18*, pages 57–73. Springer, 2016.

[SRVZ23] Florian Sextl, Adam Rogalewicz, Tomáš Vojnar, and Florian Zuleger. Sound One-Phase Shape Analysis with Biabduction. *arXiv preprint arXiv:2307.06346*, 2023.

[TLDC13] Minh-Thai Trinh, Quang Loc Le, Cristina David, and Wei-Ngan Chin. Bi-abduction with pure properties for specification inference. In *Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings 11*, pages 107–123. Springer, 2013.

[YLB$^+$08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code.

In *International Conference on Computer Aided Verification*, pages 385–398. Springer, 2008.