Partition Memory Models for Program Analysis

by

Wei Wang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Mathematics

New York University

January 2016

Professor Clark Barrett

To my parents, with love.

Acknowledgements

I would like to thank Professors Clark Barrett and Thomas Wies, my supervisors. I am grateful for their motivating guidance and support. This work could not have been done without them. I would also like to thank Professor Benjamin Goldberg, Franjo Ivancic and Daniel Schwartz-Narbonne for being my thesis committee. Finally, I would like to leave the remaining space in memory of Professor Amir Pnueli and Morgan Deters.

Abstract

Scalability is a key challenge in static program analyses based on solvers for Satisfiability Modulo Theories (SMT). For imperative languages like C, the approach taken for modeling memory can play a significant role in scalability. The main theme of this thesis is using *partitioned* memory models to divide up memory based on the alias information derived from a points-to analysis.

First, a general analysis framework based on memory partitioning is presented. It incorporates a points-to analysis as a preprocessing step to determine a conservative approximation of which areas of memory may alias or overlap and splits the memory into distinct arrays for each of these areas.

Then we propose a new *cell-based* field-sensitive points-to analysis, which is an extension of Steensgaard's unification-based algorithms. A *cell* is a unit of access with scalar or record type. Arrays and dynamically memory allocations are viewed as a collection of cells. We show how our points-to analysis yields more precise alias information for programs with complex heap data structures.

Our work is implemented in Cascade, a static analysis framework for C programs. It replaces the former flat memory model that models the memory as a single array of bytes. We show that the partitioned memory models achieve better scalability within Cascade, and the cell-based memory model, in particular, improves the performance significantly, making Cascade a state-of-the-art C analyzer.

Contents

List of Figures

List of Tables

Introduction

Solvers for Satisfiability Modulo Theories (SMT) are widely used as back ends for program analysis and verification tools. In a typical application, portions of a program's source code together with one or more desired properties are translated into formulas which are then checked for satisfiability by an SMT solver. A key challenge in many of these applications is scalability: for larger programs, the solver often fails to report an answer within a reasonable amount of time because the generated formula is too complex. Thus, one key objective in program analysis and verification research is finding ways to reduce the complexity of SMT formulas arising from program analysis.

The task is harder for programs written in C-like imperative languages featuring pointers and pointer arithmetic. The semantics requires an adequate modeling of the memory states. The theory of arrays is often used for memory modeling, with the operations read and write modeling the memory loads and stores. A natural idea is to model the memory as a single array of bytes (the flat model). This model can accurately capture the low-level constructs and operations of C-like languages including union types, type casts and pointer arithmetic. This model implicitly assumes that any two symbolic memory locations can alias or overlap, even for distinct variables and successive calls to malloc. Disjointness constraints must be introduced to guarantee the isolation of such non-overlapping locations. However, with disjointness constraints for every pair of distinct locations, the size of the generated formula grows quadratically and quickly becomes a bottleneck for scalability. Another issue affecting scalability is the byte-level reasoning required in the flat model. All the primitive values are broken into a sequence of bytes, and the corresponding memory loads or stores are decomposed into repetitive operations of array reads or writes. This increase in the number of array accesses makes the SMT formula even more complex.

The Burstall model has been proposed to improve memory modeling for higher-level imperative languages such as Java. This model splits the memory into multiple arrays according to types, making the assumption that pointers with different types will never alias. In addition to the common scalar types, each record field is also represented as a unique type. This simplifies the verification conditions by eliminating the need for disjointness constraints between pointers with distinct types. Another assumption of the model is type-safe memory access: the type with which data is stored and the type with which it is read back are guaranteed to be compatible. Based on this assumption, the element type of each memory array can be modeled using n bytes, where n is the byte size of the corresponding data type. Every memory load or store can then be represented by a single array operation, which again reduces the complexity of the resulting SMT formula. Unfortunately, this model is only suitable for strongly-typed languages. For C-like languages, the access types of memory can be easily altered via type casting or union types, and aliasing or overlapping between distinct memory areas can be arbitrarily introduced via pointer arithmetic.

One objective of this dissertation is to propose an alternative memory model for C-like languages that provides much of the efficiency of the Burstall model without losing the accuracy of the flat model. One thing learned from the Burstall model is that memory partitioning is an effective way to eliminate "uninteresting" disjointness constraints by dividing memory areas guaranteed not to be aliased into distinct memory arrays. In other words, as long as the aliased areas are allocated into one array, the partitioning should be accurate. The problem of computing the exact aliasing relations with the presence of pointers is known to be NP-hard, but a wide body of work exsits on inferring a conservative approximation. By leveraging this work, we can effectively partition memory even for C-like languages.

Contribution

The main contributions of the dissertation are to introduce the Cascade verification framework, to develop a series of partitioned memory models and especially, to introduce a novel cell-based points-to analysis that is capable of computing more precise alias information for programs with complex heap data structures, thus deriving a finer memory partition. In more details, the contributions are as follows.

The Cascade Verification Framework. The work of this thesis is implemented in Cascade, presented in chapter 2. Cascade is a static analysis platform for C, which uses bounded model checking to generate verification conditions and checks them using an SMT solver. Within the framework, the memory state is viewed as a collection of sub-states, one for each distinct alias groups. An alias-analysis module serves as a preprocessor, performing a points-to analysis on the whole program in order to discover those alias groups. This framework also provides a scalable way to analyze safety properties of programs (i.e., the absence of runtime errors such as null or dangling pointer dereferences).

Partitioned Memory Models. The alias analysis module uses a points-to analysis that attempts to construct a single points-to graph for the entire program. This graph provides a compact representation of the alias relationships. The points-to analyses can be categorized into the unification-based approach and the inclusion-based approach. In the former, each alias group points-to at most one other group; while in the later, multiple points-to edges may exist from one group to others. When interpretating the pointer dereference in the semantics, if the alias group of the pointer has multiple points-to edges, it is impossible to determine the exact points-to alias group. Therefore, we choose the unification-based points-to algorithms (Steensgaard's analysis). Both Steensgaard's original analysis and Steensgaard's field-sensitive analysis are utilized to build the field-insensitive partitioned memory model and the field-sensitive one, where the second yields more fine-grained partitions when dealing with record types (e.g. structs in C).

Cell-based Points-to Analysis. For programs with complex heap data structures, Steensgaard's field-sensitive algorithm is rather coarse in that it only distinguishes fields in static variables of record type while collapsing dynamically allocated data structures and arrays into a single alias group. To address this issue, a new cell-based points-to analysis is developed in chapter 3. A cell is a generalization of an alias group that represents a unit of access with either scalar type or a composite data type (like record or union). Cells for composite types contain inner cells representing their inner fields. Arrays are handled by merging all of the cells associated with the array's elements into a single cell. Note that if the array elements are records, then this allows arrays of records to be treated more precisely, with a separate inner cell for each record field. Data structures allocated on the heap also benefit from more precise reasoning.

Related Work

In the last decade, a variety of SAT/SMT-based automatic verifiers for C programs have been developed, such as bounded model checkers (CBMC [24], ESBMC [33], LLBMC [17], LAV [41], Corral [28] and Cascade), symbolic execution tools (KLEE [9]), and modular verifiers (VCC [11], HAVOC, and Frama-C [13]. In most cases, these tools use either flat memory models (e.g., CBMC, LLBMC, ESBMC, LLBMC, LAV, KLEE and early versions of VCC), or Burstall-style memory models (e.g., Corral). As mentioned above, users for these tools have to choose between scalability and precision in handling type-unsafe operations. Several alternative models have been proposed to achieve both.

Cohen *et al.* introduced a typed memory model for the untyped C memory [12]. This model maintains a set of valid pointers with disjoint memory locations and restricts memory accesses only to them. Special code annotation commands called split and join are introduced to switch between a typed mode and a flat mode for type casting and pointer arithmetic operations. The additional disjointness axioms are introduced for the mode switching. The axiomatization, however, imposes an extra burden on the SMT solver. Böhme *et al.* use a variant of the VCC model but few details are given [7].

Rakamarić *et al.* propose a variant of the Burstall model [35]. It employs a type unification strategy for type-unsafe operations. This optimization, however, is too coarse to handle code with even mild use of pointer casting, as the memory model will quickly degrade into the flat model.

Frama-C develops several memory models (Hoare, typed, and flat) at various abstraction levels. As an optimization strategy, Frama-C mixes the Hoare model and flat model by categorizing variables into two classes: logical variables and pointer variables. The Hoare model is used to handle the logical variables and the flat model manages the pointer variables. This strategy is similar to our partitioned model. However, our partitioned model provides more fine-grained partitions for the pointer variables.

CBMC and ESBMC use an object-based memory model. Similar to the partitioned model, it uses a static analysis to approximate for each pointer variable the set of data objects to which it might point at runtime. The data objects are assigned distinct numbers to mechanize the disjointness. However, this model is not field-sensitive, nor does it support precise reasoning over complex heap data structures.

A very large body of work concerns points-to analyses for C. We refer the reader to the survey by Hind [22]. Field-sensitive pointer analysis is specifically covered only by a relatively small subset of this work. Yong *et al.* propose a framework covering a spectrum of analyses from complete field-insensitivity through various levels of field-sensitivity [43]. Pearce *et al.* present an instance of the framework with a so-called inclusion-based approach [34]. In some sense, our analysis is also an instance of this framework but with a unification-based approach. The main difference is that our analysis is further extended to arrays and dynamically allocated regions which are not addressed in the Yong *et al.* framework.

Normally, pointer analyses are used to provide pointer information for client analyses: Mod/Ref analysis, live variable analysis, reaching definitions analysis, dependence analysis, and conditional constant propagation and unreachable code identification. Here we explore a new client – memory partitioning, and present the general analysis framework.

Outline

This dissertation is split into three parts. Chapter 1 contains background information. It reviews SMT-based program verification, symbolic execution, bounded model checking and pointer analyses. In chapter 2, we present Cascade, our static analysis platform for C. We provide the design and implementation of Cascade and the memory-partitioning verification framework. In chapter 3, we describe the cell-based points-to analysis and give the proof of soundness. In chapter 4, we provide a family of partitioned memory models over unification-based points-to analysis and present their performance with experimental results.

Chapter 1

Background and Preliminaries

1.1 Symbolic Execution

The key idea behind symbolic execution is to use symbolic values, instead of concrete data, as input values, since a single symbolic value can represent a large, potentially infinite number of concrete values [10]. During the execution, the program state is encoded as a pair $\langle \sigma, pc \rangle$, consisting of a symbolic store σ and a path condition pc. The symbolic store is a mapping from program variables to their values represented as symbolic expressions over the input symbols. Each symbolic expression is a first-order term, i.e. a symbol, or a literal number, or an operator or a function applied to first-order terms. The path condition is a first-order logic formula that tracks the history of branch decisions, which must hold on the path being explored. At the beginning of an execution, the store σ is initialized by mapping each input parameter to a fresh symbolic value, and the path condition pc is initialized to be **true**. An evaluation function $Eval(\sigma, e)$ is introduced to evaluate an expression e into a symbolic value according the current store σ. For a program with heap manipulations, a symbolic heap may be introduced as a state component that maps from locations to values. The state is updated during the course of symbolic execution.

At every assignment `v = e`, the execution updates the value of `v` in the symbolic store. Let $\langle \sigma, pc \rangle$ and $\langle \sigma', pc \rangle$ represent the pre- and post-state, where $\sigma' := \sigma[v \mapsto Eval(\sigma, \mathtt{e})]$. The conditional statement `if (e) then S1 else S2` introduces a conditional branch. The path-wise execution needs to decide which branch to select. Let $\langle \sigma, pc \rangle$ be the state at the branch point, and suppose the conditional expression `e` is evaluated as $Eval(\sigma, \mathtt{e})$. If the "then" branch is chosen, then the path condition is updated to $pc \wedge Eval(\sigma, \mathtt{e})$; otherwise, the path condition of the "else" branch is updated to $pc \wedge \neg Eval(\sigma, \mathtt{e})$. In some implementations, at the branch point, the backend constraint solver is called to check satisfiability of both path conditions and follows the path whose condition is satisfiable. If both are not satisfiable, the execution terminates. On the other hand, if both are satisfiable, again, it needs to choose one of them to continue, leaving the other for a later round.

Symbolic execution of code containing loops or recursion may result in an infinite number of paths if the termination condition for the loop or recursion is symbolic. In practice, one could either introduce loop invariants and method contracts or put a limit on the number of paths, loop iterations, or exploration depth.

During the execution, with the constraint solver, we can check if a program point is reachable by checking the satisfiability of the path condition. We can also check if a given property p, encoded as first-order formula, holds at a program point by querying the validity of the formula $pc \implies p$. Therefore, the constraint solver is the core module of the whole process. In fact, although the idea of symbolic execution was introduced more than three decades ago, it became practical only recently as a result of significant advances in the SAT and SMT solvers.

The key challenge of this technique is path explosion: the number of possible paths is usually exponential in the number of branches in the code. A standard solution in static analysis is path merging, which is to introduce a fresh state that merges all the states corresponding to different branches at the join point [26]. Techniques such as ESP [14] and trace partitioning [30] over-approximate the states of some branches; however, it is also the source of false positives. A precise alternative relies on using ITE-expressions that simply combine the information from all the incoming branches.

For a conditional statement `if (e) then S1 else S2`, let $\langle \sigma_0, pc_0 \rangle$ be the state at the branch point and $\langle \sigma_1, pc_1 \rangle$, $\langle \sigma_2, pc_2 \rangle$ the states of the "then" and "else" branches, where $pc_1 = pc \wedge \sigma_0(e)$ and $pc_2 = pc \wedge \neg\sigma_0(e)$. For the merged state $\langle \sigma_3, pc_3 \rangle$ at the join point, the path condition pc_3 is $pc_1 \vee pc_2$ which is equivalent to pc, and $\forall v \in \mathtt{dom}(\sigma_3)\ .\ \sigma_3(v) = \mathtt{ite}(\sigma_0(e), \sigma_1(v), \sigma_2(v))$. In this way, the execution is no longer path-based, while multiple (not all) of the possible paths of the whole program are encoded into one first-order formula that is passed directly to the constraint solver. The problem of path explosion is reduced; however, a huge number of disjunctions are introduced into the formula, which can be hard to reason about for SAT and SMT solvers.

1.2 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) aims to check the satisfiability of first order logical formulas over one or more background theories. Solvers for Satisfiability Modulo Theories play a central role in program analysis and verification as the semantics of most program statements are easily modeled using theories supported by most SMT solvers [16]. Here, we give a brief description of theories referenced in the later chapters.

1.2.1 Theory of Arrays

The theory of arrays was introduced by John McCarthy [31] in 1962 and is widely used for modeling memory. It is parameterized by the index sort T_I and element sort T_E, with T_A a shorthand for $T_I \rightarrow T_E$, i.e., the set of functions that map an element of T_I to an element in T_E. There are two operations on arrays: $\mathsf{read} : T_A \times T_I \rightarrow T_E$ and $\mathsf{write} : T_A \times T_I \times T_E \rightarrow T_A$. The write function is used to store an element in an array, and the read function is used to retrieve an element from an array [25]. The main axiom used to defined the meaning of the two operators is the "read-over-write axiom": after the value e has been written into array a at index i, the value of this array at index i is e. The value at any index $j \neq i$ remaints unchanged after the write operation:

$$\forall a \in T_A \,.\, \forall i, j \in T_I \,.\, \forall e \in T_E \,.\, (i = j \implies \mathsf{read}(\mathsf{write}(a, i, e), j) = e) \land$$
$$(i \neq j \implies \mathsf{read}(\mathsf{write}(a, i, v), j) = \mathsf{read}(a, j))$$

1.2.2 Theory of Fixed-Size Bit-vectors

The theory of bit-vectors captures the semantics of modular arithmetic and is capable of discovering bugs caused by overflows. A bit-vector is a sequence of bits. The size of a bit-vector is the length of this sequence. It could be used to encode both positive and negative numbers (with signed bit vectors), or just encode positive numbers (with unsigned bit vectors) [25]. The theory of fixed-size bit-vectors is made of variables and constants of arbitrary but fixed sizes, and functions and predicates operating on them. The operators include extraction, concatenation, bit-wise Boolean operations, and arithmetic operations. For the result of arithmetic operations, if the number of bits exceeds the given size, the additional bits are discarded. This is a good match with the modular arithmetic of machine languages.

1.3 Points-to Analysis

The goal of points-to analysis is to determine the set of locations that a pointer may point-to at runtime. The result of the analysis is a points-to graph, whose nodes are sets of program expressions and whose edges represent the may points-to relation. The graph provides a compact representation of alias information: two pointers are aliased if they point-to the same node.

The algorithms of points-to analysis can be categorized into the unification-based approach and the inclusion-based approach. The unification-based approach was proposed by Steensgaard [40]. The key idea is that for a pointer assignment `p = q`, the points-to set of `p` and the points-to set of `q` are required be equal. It is implemented with the union-find algorithm, which is known for its almost linear time cost in terms of the program size. The inclusion-based approach was proposed by Andersen [1], with the worst-time complexity $O(n^3)$. For an assignment `p = q`, the points-to set of `q` is required to be a subset of the points-to set of `p`. Compared with the equality constraint, this inclusion constraint can yield more precise results.

There are several other dimensions that can be used to classify points-to analyses, such as flow-sensitivity, context-sensitivity and field-sensitivity [22]. A flow-insensitive analysis ignores the control flow information and computes the points-to graph of the whole program, whereas a flow-sensitive analysis builds the points-to graph at each program point. A context-insensitive analysis merges the points-to relations for all the calling contexts of a method, whereas a context-sensitive analysis separates them for different calling contexts. The key difference between field-sensitive and insensitive analyses is whether to distinguish the components within record types as separate objects or to collapse them into one object. It is difficult to apply field-sensitive analyses to weakly-typed languages like C/C++. Note that both Andersen's and Steensgaard's analyses are flow-, context- and field-insensitive.

Chapter 2

Analysis Framework with Memory Partitioning

In this chapter, we introduce an analysis framework based on memory partitioning. This framework is implemented in Cascade, a static analysis tool for C programs. Section 2.1 reviews the features and workflow of Cascade. Then Section 2.2 gives a overview of the analysis framework using simple examples. Section 2.3 provides a formal description of symbolic execution within the framework. Finally, Section 2.4 describes the encoding of memory constraints.

2.1 Cascade

Cascade is an open-source tool developed at New York University for automatically reasoning about C programs. An initial prototype of the system was described in [37]. In this chapter, we describe the latest version Cascade 2.0 [42], a from-scratch reimplementation which provides a number of new features, including support for nearly all of C (with the exception of floating point) and a new back-end theorem prover interface supporting both CVC4 [2] and Z3 [15].

Cascade supports arbitrary user assertions, including reachability of labels in the C-code. Furthermore, it can detect bugs related to memory safety, including invalid memory accesses, invalid memory frees, and memory leaks. As a bounded model checker, Cascade relies on loop unrolling and function inlining, and thus only ensures the correctness for programs for which this can be bounded. With unbounded loops or recursive functions, it does not provide sound results.

Cascade 2.0 is implemented in Java. The overall framework is illustrated in Figure. 2.1. The C front-end converts a target program into an abstract syntax tree using a parser built using the xtc parser generator [19]. The core module uses symbolic execution [6, 8, 23] to build verification conditions as SMT formulas. Currently, it takes the approach of simple forward execution.

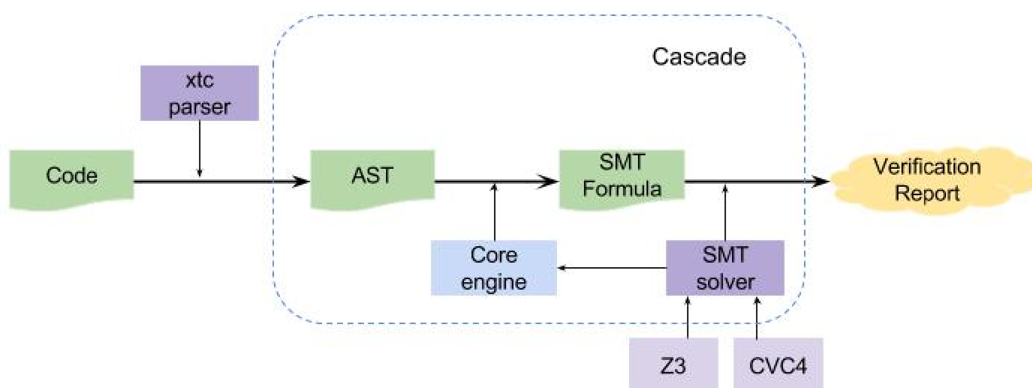

Figure 2.1: The framework of Cascade

The workflow of the core engine is presented in Fig. 2.2. In the preprocessing module, a *unification-based* points-to analysis is performed for each function in the

C program (before function-inlining or loop-unrolling). All the alias groups and the points-to relations among them are discovered here. The partitioned memory model is built based on the alias information generated in the preprocessing step.

After preprocessing, the abstract syntax tree is translated into a loop-free and call-free control flow graph via function-inlining and loop-unrolling. The symbolic execution is then performed on the control flow graph to generate the verification conditions as SMT formulas. For function inlining, Cascade takes a given depth bound, inlines functions up to that bound, and then continues running if all the functions are fully inlined, otherwise, it stops and returns UNKNOWN. For loop unrolling, Cascade uses successively larger unrolls until it reaches a given unroll limit. After each loop unrolling, it asks the SMT solver if it has unrolled enough, and stops there if it has. Cascade only ensures the correctness of programs for which this approach succeeds.

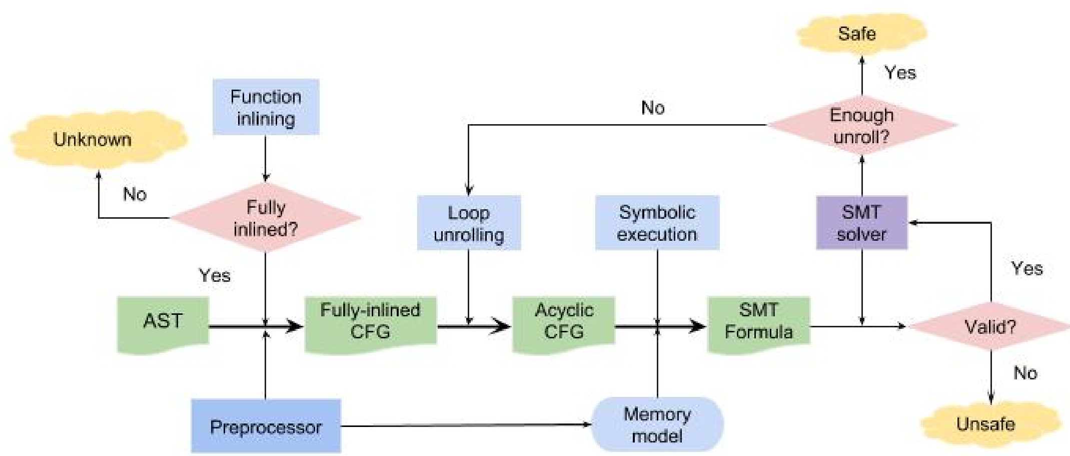

Figure 2.2: The workflow of the core engine

2.2 Overview of the Analysis Framework

In this section, we try to present an informal overview of our framework for symbolic execution with memory partitioning. Before getting to the details, let us review the *classical* framework of symbolic execution (without memory partitioning). The program inputs are represented as symbolic values, instead of concrete values, and the program variables are represented as symbolic expressions over the symbolic input values [10]. In order to support the address-of (&) operation in C, a *symbolic store* ϵ is kept to map program variables to their left-values (memory addresses) rather than right-values (symbolic expressions). In C programs, one can apply this operator to any variable, generating a pointer value that can be used to update the value of the variable. Note that the left-values in ϵ are represented as *unique* symbolic variables, and ϵ does not change during the course of an execution.

Symbolic execution maintains a *symbolic state* $\sigma = \langle pc, m \rangle$, where pc is a symbolic path condition (a first-order formula), and m is a memory state mapping from memory addresses to values. At the beginning of an execution, m is a fresh array variable and pc is initialized to True. Both can be changed as the program executes.

The key idea behind the framework with memory partitioning is to use separated memory states for distinct memory partitions (created for different alias groups computed at the preprocessing step). Each memory partition has a unique identifier that is the identifier of the corresponding alias group. Therefore, the symbolic state contains a symbolic path condition pc, and a memory state map ρ, which maps the identifiers of memory partitions (alias groups) to their memory states, denoted as $\sigma = \langle pc, \rho \rangle$. At the beginning of an execution, ρ is initialized as an empty map (denoted as $\emptyset$). If a new memory partition with identifier k, is encountered during the execution, the memory state map ρ is expanded by mapping k to a fresh memory state m_k, a new array variable.

Note that in a *field-sensitive* points-to analysis, an alias group of a program expression with record or union type may contain inner alias groups representing the nested fields, and thus the corresponding memory partition may also contain inner partitions. In this case, when a new memory partition is added into the memory state map ρ, its inner partitions are also added with fresh memory states.

For convenience, a few helper functions are introduced to simulate the queries to the preprocessor for the alias group of a program expression, the points-to alias group and the inner groups of a given alias group:

- Given an expression e, function $\mathsf{partition}(e)$ returns the alias group identifier of e.

- For an alias group with identifier k, function $\mathsf{ptsto}(k)$ returns the identifier of its points-to alias group. Because the preprocessor is built on a unification-based points-to analysis, each alias group can point to no more than one alias group.

- For an alias group with identifier k, function $\mathsf{partitions}(k)$ returns the set of identifiers, consisting of k and the identifiers of its inner groups. For a *field-insensitive* points-to analysis, $\mathsf{partitions}(k) = \{k\}$.

Motivating Example. In the following, we illustrate the symbolic execution with memory partitioning via the sample code in Fig. 2.3. The function `ldv_malloc` is a wrapper function for `malloc` which ensures that if the input `size` is not positive, the returned pointer is 0 (null pointer).

```
void * ldv_malloc(long size) {
  if (size <= 0) {
    return 0;
  } else {
    return malloc(size);
  }
}
```

Figure 2.3: A wrapper function for `malloc`

State Initialization. In the symbolic store ϵ, all variables are bound with their left-values. In the sample code, the input parameter `size` is viewed as a local variable. Besides, for each function with non-`void` return type, an auxiliary return variable is created and any return statement within the function is viewed an assignment to the return variable. For function `ldv_malloc`, `ret_ldv_malloc` is the return variable. Thus, within the symbolic store ϵ, we have $\epsilon(\texttt{size}) = \mathsf{size}$ and $\epsilon(\texttt{ret\_ldv\_malloc}) = \mathsf{ret\_ldv\_malloc}$, where the left-values are denoted with sans serif font. The initial program state is $\sigma_0 = (\mathsf{True}, \emptyset)$.

Variable Declaration. When entering a function, its input parameters and return variable are treated as newly declared variables. For each variable declaration, a fresh memory state is created for its memory partition if it is not included in the current memory map. In the sample code, two variables `size` and `ret_ldv_malloc` are declared. Let $k_1 = \mathsf{partition}(\texttt{size})$, $k_2 = \mathsf{partition}(\texttt{ret\_ldv\_malloc})$, where $\mathsf{partitions}(k_1) = \{k_1\}$, $\mathsf{partitions}(k_2) = \{k_2\}$. Then two fresh memory states m_{k_1} and m_{k_2} are created. The program state is then updated as

$$\sigma_1 = (\mathsf{True},\ \ \{k_1 \mapsto m_{k_1}, k_2 \mapsto m_{k_2}\})$$

The value of variable `size` is a dereference of the memory m_{k_1} via its left-value, represented as $\mathsf{read}(m_{k_1}, \mathsf{size})$. Because m_{k_1} is a fresh array variable, the value of `size` is non-deterministic, representing all possible concrete input values.

State Branching. The function body of `ldv_malloc` consists of a single conditional statement. After entering this function, the execution is split into two branches with the condition expression `size <= 0`. In σ_1, the condition expression is evaluated as $\mathsf{read}(m_{k_1}, \mathsf{size}) \leq 0$. The path condition pc is updated to $\mathsf{read}(m_{k_1}, \mathsf{size}) \leq 0$ ("then" branch), and $\mathsf{read}(m_{k_1}, \mathsf{size}) > 0$ ("else" branch). Correspondingly, the program state is updated as σ_2 ("then" branch) and σ_3 ("else" branch):

$$\sigma_2 = (\mathsf{read}(m_{k_1}, \mathsf{size}) \leq 0, \ \ \{k_1 \mapsto m_{k_1}, k_2 \mapsto m_{k_2}\})$$
$$\sigma_3 = (\mathsf{read}(m_{k_1}, \mathsf{size}) > 0, \ \ \{k_1 \mapsto m_{k_1}, k_2 \mapsto m_{k_2}\})$$

Assignment. Both "then" and "else" branches have a return statement. As discussed earlier, the return statement is viewed as assignment to the return variable `ret_ldv_malloc`. In the "then" branch, `ret_ldv_malloc` is assigned to 0 and σ_2 is updated as

$$\sigma_{21} = (\mathsf{read}(m_{k_1}, \mathsf{size}) \leq 0, \ \ \{k_1 \mapsto m_{k_1}, k_2 \mapsto \mathsf{write}(m_{k_2}, \mathsf{ret\_ldv\_malloc}, 0)\})$$

Memory Allocation. For memory allocation, a fresh memory region is generated, and a fresh address variable is created to represent the base address of the newly-allocated region. In the function `ldv_malloc`, a fresh memory region with size `size` is allocated in the "else" branch, whose base address is represented by a fresh address variable region. The variable region is then assigned to the return variable `ret_ldv_malloc`. So the memory state m_{k_2} is updated to

$\mathsf{write}(m_{k_2}, \mathsf{ret\_ldv\_malloc}, \mathsf{region})$.

Moreover, a fresh memory state is created for the newly allocated region. Since the region is pointed to by `ret_ldv_malloc`, the alias group of the region must be pointed to by the alias group of `ret_ldv_malloc`. Let $k_3 = \mathsf{ptsto}(k_2)$, i.e., $\mathsf{partitions}(k_3) = \{k_3\}$. Suppose $k_3 \neq k_2$ and $k_3 \neq k_1$, state σ_3 is updated as

$$\sigma_{31} = \left(\mathsf{read}(m_{k_1}, \mathsf{size}) > 0, \quad \left\{ \begin{array}{lcl} k_1 & \mapsto & m_{k_1}, \\ k_2 & \mapsto & \mathsf{write}(m_{k_2}, \mathsf{ret\_ldv\_malloc}, \mathsf{region}), \\ k_3 & \mapsto & m_{k_3} \end{array} \right\} \right)$$

State Merging. At the join point of both branches, a fresh state σ_4 is created to merge the branch states σ_{21} and σ_{31}. In σ_4, the merged path condition is a disjunction of the path conditions in the branch states, where $\mathsf{read}(m_{k_1}, \mathsf{size}) \leq 0 \vee \mathsf{read}(m_{k_1}, \mathsf{size}) > 0 = \mathsf{True}$.

When merging memory state maps, the memory state are merged using the ITE-expressions. If there is any memory partition with identifier k not tracked in some branches, we use the initial memory state m_k as a default state without any written value. In the "then" branch, the memory partition with identifier k_3 is missing and its default memory state m_{k_3} is used. Therefore, σ_4 is as

$$\sigma_4 = \left(\mathsf{True}, \quad \left\{ \begin{array}{lcl} k_1 & \mapsto & m_{k_1}, \\ k_2 & \mapsto & \mathtt{ite}(\mathsf{read}(m_{k_1}, \mathsf{size}) \leq 0, \\ & & \quad \mathsf{write}(m_{k_2}, \mathsf{ret\_ldv\_malloc}, 0), \\ & & \quad \mathsf{write}(m_{k_2}, \mathsf{ret\_ldv\_malloc}, \mathsf{region})), \\ k_3 & \mapsto & m_{k_3} \end{array} \right\} \right)$$

Property Checking. When checking a given property at a given program point, we first evaluate the property into a symbolic formula according to the current program state and then check if the formula is implied by the conjunction of the path condition and the disjointness constraint. Note that the disjointness constraint of the current program state is a conjunction of the disjointness constraints of all memory partitions.

One desired property that must hold at the exit point of function `ldv_malloc` is that the returned pointer must be 0 if `size` is not positive, represented as:

$$\texttt{size} \leq 0 \implies \texttt{ret\_ldv\_malloc} = 0$$

According to the exit state σ_4, the above formula is evaluated as:

$$\begin{array}{l} \mathsf{read}(m_{k_1}, \mathsf{size}) \leq 0 \implies \\ \qquad \mathsf{read}\left(\texttt{ite}\left(\begin{array}{l} \mathsf{read}(m_{k_1}, \mathsf{size}) \leq 0, \\ \quad \mathsf{write}(m_{k_2}, \mathsf{ret\_ldv\_malloc}, 0), \\ \quad \mathsf{write}(m_{k_2}, \mathsf{ret\_ldv\_malloc}, \mathsf{region}) \end{array}\right), \mathsf{ret\_ldv\_malloc}\right) = 0 \end{array}$$

which can be further simplified to True, so the property holds. Note that the path condition of σ_4 is True, and the disjointness constraint is also True, because the memory partitions k_1, k_2 and k_3 each contain only one memory region or variable.

2.3 Formalization

In this section, we formalize the static symbolic execution with memory partitioning. The formalization is motivated by the work of Schwartz *et al.* [36]. It provides

$$
\begin{array}{llr}
\mathtt{f} ::= \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m & & \text{fields} \\
\mathtt{t} ::= \mathtt{uint8} \mid \mathtt{int8} \mid \ldots \mid \mathtt{int64} \mid \mathtt{ptr} & & \text{scalar types} \\
\quad \mid \mathtt{struct}\{\, \mathtt{t}_1\mathtt{f}_1; \ldots \mathtt{t}_n\mathtt{f}_n;\, \}\,\mathtt{S} & & \text{record types} \\
\quad \mid \mathtt{union}\{\, \mathtt{t}_1\mathtt{f}_1; \ldots \mathtt{t}_n\mathtt{f}_n;\, \}\,\mathtt{U} & & \text{union types} \\
 & & \\
s \ ::= \mathtt{declare}\ \mathtt{x} \ \mid\ e_1 =_{\mathtt{t}} e_2 & & \mathtt{t}\ \text{is scalar} \\
\quad \mid e_1 = \mathtt{malloc}(e_2) \ \mid\ \mathtt{free}\ e & & \\
\quad \mid \mathtt{assume}\ e \ \mid\ \mathtt{assert}\ e & & \\
\quad \mid \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2 \ \mid\ s_1; s_2 & &
\end{array}
\qquad
\begin{array}{llr}
e \ ::= n & & \text{constant} \\
\quad \mid \mathtt{x} & & \text{variable} \\
\quad \mid e.\mathtt{f} & & \text{field selection} \\
\quad \mid \&\mathtt{x} & & \text{address of} \\
\quad \mid *e & & \text{dereference} \\
\quad \mid (\mathtt{t})\ e & & \text{cast, } \mathtt{t} \text{ is scalar} \\
\quad \mid op_u\ e & & op_u \in \{-, !, \sim\} \\
\quad \mid e_1\ op_b\ e_2 & & op_b \in \{+, -, *, \ldots\}
\end{array}
$$

Figure 2.4: Language syntax

a concise and precise way to define the analysis framework. Section 2.3.1 presents a simple C-like language that serves as the target of our analysis. Section 2.3.2 defines the program state and the sub-state of memory partitions. Section 2.3.3 gives the semantics of expressions and statements.

2.3.1 Syntax

Fig. 2.4 lists the types and syntax of a simple C-like language that captures the core features including pointer arithmetic, structs and heap manipulations. We assume the C program has been processed into a sequence of statements. These statements do not include iterations and function calls (i.e., the program has been preprocessed into a loop-free and call-free fragment.

In the type system, all pointers use a single type denoted by `ptr`. Pointers and integer types are viewed as scalar types, distinguished from composite types (structs, unions and arrays). Function types and floating point types are not considered, since functions are inlined and floating points are not supported here. For a given type $\mathtt{t}$, $|\mathtt{t}|$ is the byte-size of $\mathtt{t}$.

For brevity, the type inference process is omitted. We assume each expression is already tagged with a type. For a given expression e, function $\mathsf{typeof}(e)$ returns the associated type, and function $\mathsf{sizeof}(e)$ returns the byte-size of $\mathsf{typeof}(e)$. Given a struct or union type $\mathtt{t}$ and a field identifier $\mathtt{f}$, function $\mathsf{offsetof}(\mathtt{t}, \mathtt{f})$ returns the offset value in bytes of the field $\mathtt{f}$ within $\mathtt{t}$.

Let $\mathbb{X}$ denote a fixed set of variables, and let $\mathbb{V}$ denote the set of values. For any type $\mathtt{t}$, $\mathbb{V}_{\mathtt{t}} \subseteq \mathbb{V}$ is the set of values with type $\mathtt{t}$, for example, $\mathbb{V}_{\mathtt{ptr}}$ is the set of addresses. For type casting, we introduce a function $\mathsf{convert} : \text{type} \times \mathbb{V} \to \mathbb{V}$. For $a \in \mathbb{V}$, $\mathsf{convert}(\mathtt{t}, a)$ is the result of casting the value a to type $\mathtt{t}$.

2.3.2 Program State

Let us denote by $\mathbb{P}$ the set of memory partition identifiers, then function $\mathsf{partition}$, ptsto and $\mathsf{partitions}$ are formalized as:

- $\mathsf{partition} : \text{expr} \to \mathbb{P}$ gives the memory partition of an expression;
- $\mathsf{ptsto} : \mathbb{P} \to \mathbb{P}$ gives points-to partition of a given partition;
- $\mathsf{partitions} : \mathbb{P} \to \mathcal{P}(\mathbb{P})$ gives a set of contained memory partitions of a given partition, which includes the given partition and its inner partitions. In the field-insensitive points-to analysis, $\forall k \in \mathbb{P} \,.\, \mathsf{partitions}(k) = \{k\}$.

State of Memory Partition. Recall that the memory state map maps partition identifiers to states of memory partitions. A state of memory partition is a pair of memory state and *memory constraint*. We let $\mathbb{M}$ denote the set of memory states. The memory read/write takes a size k as a parameter that reads/writes exactly k bytes:

- $\mathsf{read} : \mathbb{M} \times \mathbb{V}_{\mathtt{ptr}} \times \mathbb{N} \to \mathbb{V}$;

- $\mathsf{write} : \mathbb{M} \times \mathbb{V}_{\mathtt{ptr}} \times \mathbb{N} \times \mathbb{V} \to \mathbb{M}$.

Let Φ denote the set of memory constraints describing memory-related properties. The encoding and transition of memory constraints are discussed in the section 2.4. The state of memory partition is (m, ϕ), where $m \in \mathbb{M}$ is the current memory state and $\phi \in \Phi$ is the associated memory constraint. The set of memory partition states is denoted as $\mathbb{M} \times \Phi$.

Program State. The *state* of the program is required as a tuple $\sigma = \langle \epsilon, \rho, \mu, pc \rangle$:

- $\epsilon : \mathbb{X} \to \mathbb{V}_{\mathtt{ptr}}$ is a partial function from variable identifiers to left-values;
- $\rho : \mathbb{P} \to \mathbb{M} \times \Phi$ is a partial function from memory partition identifiers to their states;
- μ is a formula capturing any assumptions made by `assume` statements;
- pc is the path condition.

In the following, we use the components of specific states by using the appropriate letter subscripted by the state. Thus ρ_σ represents the memory partition state mapping for state σ. The notation $\sigma[\rho \coloneqq \rho']$ represents the state that is identical to σ except that the component ρ has been replaced with a new value ρ'. The *initial* state is $\sigma_0 = \langle \epsilon, \emptyset, \mathsf{True}, \mathsf{True} \rangle$.

2.3.3 Operational Semantics

The semantics is specified using natural semantics, also known as big-step operational semantics, rather small-step semantics. This choice simplifies our work by

$$\textsc{Non-Scalar}\ \frac{\mathsf{typeof}(e) \text{ is not scalar}}{\langle \sigma, e, loc \rangle \leadsto loc}$$

$$\textsc{Scalar}\ \frac{\mathsf{typeof}(e) \text{ is scalar} \quad \mathsf{partition}(e) = k \quad \rho_\sigma(k) = \langle m_k, \phi_k \rangle}{\langle \sigma, e, loc \rangle \leadsto \mathsf{read}(m_k, loc, \mathsf{sizeof}(e))}$$

Figure 2.5: Right-value evaluation

avoiding modeling the non-determinism of the semantics of C. The semantics is defined by three judgements:

- $\langle \sigma, e \rangle \Downarrow^l loc$, left-value evaluation of expression, where loc is the left-value of expression e evaluated in state σ;
- $\langle \sigma, e \rangle \Downarrow^v a$, right-value evaluation of expression, where a is the value of expression e evaluated in state σ;
- $\langle \sigma, s \rangle \Downarrow \sigma'$, σ' is the updated state of σ after statement s.

In many contexts, left-values to become right-values for a given expression. For such cases, we introduce the notation $\langle \sigma, loc, e \rangle \leadsto a$, where σ is the current state, loc is the left-value, e is the given expression and a is the right-value. The definition of $\leadsto$ is given in Fig. 2.5. If the type of e is not scalar, as shown in the rule Non-Scalar, the right-value is the left-value. Otherwise, as shown in the rule Scalar, let the memory partition of e have identifier k and memory state m_k. Then the right-value is the value read from m_k at address loc with byte-size $\mathsf{sizeof}(e)$.

2.3.3.1 Expression Evaluation

Given an expression e and a state σ, we can determine the value of the expression in σ via the evaluation rules presented in Fig. 2.6. These expressions are free of side-effects and the program state is not changed during the evaluation.

$$\textsc{Const}\ \frac{}{\langle \sigma, c\rangle \Downarrow^v c} \qquad \textsc{Var}^l\ \frac{\epsilon_\sigma(\mathtt{v}) = loc}{\langle \sigma, \mathtt{v}\rangle \Downarrow^l loc} \qquad \textsc{Var}^v\ \frac{\langle \sigma, \mathtt{v}\rangle \Downarrow^l loc \qquad \langle \sigma, \mathtt{v}, loc\rangle \rightsquigarrow a}{\langle \sigma, \mathtt{v}\rangle \Downarrow^v a}$$

$$\textsc{FieldSel}^l\ \frac{\langle \sigma, e\rangle \Downarrow^l loc}{\langle \sigma, e.\mathtt{f}\rangle \Downarrow^l loc + \mathsf{offsetof}(\mathsf{typeof}(e), \mathtt{f})}$$

$$\textsc{FieldSel}^v\ \frac{\langle \sigma, e.\mathtt{f}\rangle \Downarrow^l loc \qquad \langle \sigma, e.\mathtt{f}, loc\rangle \rightsquigarrow a}{\langle \sigma, e.\mathtt{f}\rangle \Downarrow^v a} \qquad \textsc{Addr}\ \frac{\epsilon_\sigma(\mathtt{v}) = a}{\langle \sigma, \&\mathtt{v}\rangle \Downarrow^v a}$$

$$\textsc{Deref}^l\ \frac{\langle \sigma, e\rangle \Downarrow^v loc \qquad \langle \sigma, *e, loc\rangle \rightsquigarrow a}{\langle \sigma, *e\rangle \Downarrow^l a} \qquad \textsc{Deref}^v\ \frac{\langle \sigma, *e\rangle \Downarrow^l a}{\langle \sigma, *e\rangle \Downarrow^v a}$$

$$\textsc{Cast}\ \frac{\langle \sigma, e\rangle \Downarrow^v a}{\langle \sigma, (\mathtt{t})\ e\rangle \Downarrow^v \mathsf{convert}(\mathtt{t}, a)} \qquad \textsc{Unary-Op}\ \frac{\langle \sigma, e\rangle \Downarrow^v a}{\langle \sigma, op_u\ e\rangle \Downarrow^v op_u\ a}$$

$$\textsc{Binary-Op}\ \frac{\langle \sigma, e_1\rangle \Downarrow^v a_1 \qquad \langle \sigma, e_2\rangle \Downarrow^v a_2}{\langle \sigma, e_1\ op_b\ e_2\rangle \Downarrow^v a_1\ op_b\ a_2}$$

Figure 2.6: Expression evaluation

2.3.3.2 Statement Semantics

The semantics of statements affects the transition of program states, denoted via the judgement $\langle \sigma, s\rangle \Downarrow \sigma'$. Note that details related to the initialization and transition of memory constraint ϕ, the second component of states, are omitted here, but are presented in section 2.4.

▶ **Variable Declaration**

$$\frac{}{\langle \sigma, \mathtt{declare\ v}\rangle \Downarrow \sigma[\rho \coloneqq \rho']}$$

Let $k_v = \mathsf{partition}(v)$, then for any $k \in \mathsf{partitions}(k_v)$,

- if $k \in \mathtt{dom}(\rho)$ and $\rho(k) = \langle m_k, \phi_k\rangle$, then $\rho'(k) = \langle m_k, \phi'_k\rangle$, where the memory constraint is updated to ϕ'_k;
- otherwise, $\rho'(k) = \langle m_k, \phi_k\rangle$, where m_k is a fresh array variable and ϕ_k is a

fresh memory constraint.

▶ **Assignment**

$$\frac{\langle \sigma, e_1 \rangle \Downarrow^l loc \qquad \langle \sigma, e_2 \rangle \Downarrow^v a \qquad \mathsf{partition}(e_1) \in \mathtt{dom}(\rho)}{\langle \sigma, e_1 =_{\mathtt{t}} e_2 \rangle \Downarrow \sigma[\rho \coloneqq \rho']}$$

Let $k = \mathsf{partition}(e_1)$ and $\rho(k) = \langle m_k, \phi_k \rangle$, then $\rho'(k) = \langle m'_k, \phi_k \rangle$ where

$$m'_k = \mathsf{write}(m_k, loc, |\mathtt{t}|, a)$$

▶ **Memory Allocation**

$$\frac{\langle \sigma, e_1 \rangle \Downarrow^l loc \qquad \langle \sigma, e_2 \rangle \Downarrow^v a \qquad \mathsf{region} \text{ is fresh} \qquad \mathsf{partition}(e_1) \in \mathtt{dom}(\rho)}{\langle \sigma, e_1 = \mathtt{malloc}(e_2) \rangle \Downarrow \sigma[\rho \coloneqq \rho', \mu \coloneqq \mu']}$$

For each allocation statement, a fresh variable $\mathsf{region} \in \mathbb{V}_{\mathtt{ptr}}$ is created to represent the base address of the newly-allocated region. Assumptions are made over region in order to ensure the memory allocation is valid: (1) the base address is not null pointer, and (2) the allocated region is within the bound of address space. Therefore,

$$\mu' \quad \equiv \quad \mu \wedge \mathsf{region} \neq 0 \wedge \mathsf{region} \leq \mathsf{region} +_{|\mathtt{ptr}|} a$$

where $+_{|\mathtt{ptr}|}$ is modular addition with modulus $2^{|\mathtt{ptr}|}$.

Let $k = \mathsf{partition}(e_1)$ and $\rho(k) = \langle m_k, \phi_k \rangle$, then $\rho'(k) = \langle m'_k, \phi_k \rangle$ where

$$m'_k = \mathsf{write}(m_k, loc, |\mathtt{ptr}|, \mathsf{region})$$

Let $k_* = \mathsf{ptsto}(k)$, the memory partition identifier of the newly allocated memory region with base address region and size a. Then for any $k' \in \mathsf{partitions}(k_*)$,

- if $k' \in \mathtt{dom}(\rho)$ and $\rho(k') = \langle m_{k'}, \phi_{k'} \rangle$, then $\rho'(k') = \langle m_{k'}, \phi'_{k'} \rangle$, where the memory constraint is updated to $\phi'_{k'}$;
- otherwise, $\rho'(k') = \langle m_{k'}, \phi_{k'} \rangle$, where $m_{k'}$ is a fresh array variable and $\phi_{k'}$ is a fresh memory constraint.

▶ **Memory Deallocation**

$$\frac{\langle \sigma, e \rangle \Downarrow^v \mathit{loc}}{\langle \sigma, \mathtt{free}\ e \rangle \Downarrow \sigma[\rho \coloneqq \rho']}$$

Let $k = \mathsf{ptsto}(\mathsf{partition}(e))$, the memory partition identifier of the memory region pointed by e. For any $k' \in \mathsf{partitions}(k)$ and $\rho(k') = \langle m_{k'}, \phi_{k'} \rangle$, then $\rho'(k') = \langle m_{k'}, \phi'_{k'} \rangle$, where the memory constraint is updated to $\phi'_{k'}$.

▶ **Assumption**

$$\frac{\langle \sigma, e \rangle \Downarrow^v a}{\langle \sigma, \mathtt{assume}\ e \rangle \Downarrow \sigma[\mu \coloneqq \mu \wedge a]}$$

For the assume statement, the assumption component of the current state σ is updated as a conjunction of μ and the result of evaluating e.

▶ **Assertion**

$$\frac{\langle \sigma, e\rangle \Downarrow^v a \qquad \mathsf{disjoint}(\rho_\sigma) \wedge \mu_\sigma \wedge \phi_\sigma \implies a}{\langle \sigma, \mathtt{assert}\ e\rangle \Downarrow \sigma}$$

For each assert statement, we check that its boolean expression is implied by the current path condition ϕ, the assumption formula μ, and the disjointness constraint $\mathsf{disjoint}(\rho_\sigma)$ whose definition is given in (2.22) in section 2.4. If the check succeeds, the execution continues; otherwise, the execution aborts.

▶ **Sequence**

$$\frac{\langle \sigma, s_1\rangle \Downarrow \sigma_1 \qquad \langle \sigma_1, s_2\rangle \Downarrow \sigma_2}{\langle \sigma, s_1; s_2\rangle \Downarrow \sigma_2}$$

▶ **Conditional**

$$\frac{\langle \sigma, e\rangle \Downarrow^v a \qquad \langle \sigma, s_1\rangle \Downarrow \sigma_1 \qquad \langle \sigma, s_2\rangle \Downarrow \sigma_2}{\langle \sigma, \mathtt{if}\ (e)\ s_1\ \mathtt{else}\ s_2\rangle \Downarrow \sigma[\mu \coloneqq \mathtt{ite}(a, \mu_{\sigma_1}, \mu_{\sigma_2}), \rho \coloneqq \rho']}$$

For any memory partition $k \in \mathtt{dom}(\rho_{\sigma_1}) \cup \mathtt{dom}(\rho_{\sigma_2})$, we consider the semantics it by following three cases:

- if $k \in \mathtt{dom}(\rho_{\sigma_1}) \cap \mathtt{dom}(\rho_{\sigma_2})$, where $\rho_{\sigma_1}(k) = \langle m_{k_1}, \phi_{k_1}\rangle$ and $\rho_{\sigma_2}(k) = \langle m_{k_2}, \phi_{k_2}\rangle$,

$$\rho'(k) = \langle \mathtt{ite}(a, m_{k_1}, m_{k_2}), \phi_{k_1} \sqcup_a \phi_{k_2}\rangle$$

  where $\phi_{k_1} \sqcup_a \phi_{k_2}$ denotes the join of memory constraints under condition a

whose definition is given in (2.23) in section 2.4;

- if $k \in \mathtt{dom}(\rho_{\sigma_1}) \setminus \mathtt{dom}(\rho_{\sigma_2})$, let $\rho_{\sigma_1}(k) = \langle m_{k_1}, \phi_{k_1} \rangle$ and

$$\rho'(k) = \langle \mathtt{ite}(a, m_{k_1}, m_k), \phi_{k_1} \sqcup_a \phi_k \rangle$$

where m_k is a fresh array variable and ϕ_k is a fresh memory constraint;

- if $k \in \mathtt{dom}(\rho_{\sigma_2}) \setminus \mathtt{dom}(\rho_{\sigma_1})$, let $\rho_{\sigma_2}(k) = \langle m_{k_2}, \phi_{k_2} \rangle$ and

$$\rho'(k) = \langle \mathtt{ite}(a, m_k, m_{k_2}), \phi_k \sqcup_a \phi_{k_2} \rangle$$

where m_k is a fresh array variable and ϕ_k is a fresh memory constraint.

2.4 Memory Constraints Encoding

In this section, we discuss the encoding of memory constraints, including the constraint of disjointness, denoting the non-overlapping of valid memory regions generated either via variable declaration or memory allocation. We also discuss memory safety checks such as valid memory access, valid frees and no memory leaks. These constraints are associated with memory partitions as a state component ϕ, and get updated during the program execution.

2.4.1 Disjointness

The *disjointness* constraint specifies that all memory blocks (generated either via variable declaration or memory allocation) are non-overlapping. In our semantics, the disjointness is applied to both allocated and de-allocated blocks, which means

```c
typedef struct S {
  int x;
  struct S *l;
  struct S *r;
} S;

S *bar(int arg) {
  S *p;
  p = malloc(sizeof(S));
  p->x = 5;
  if(arg > 0)
    p->l = malloc(sizeof(S));
  else
    p->r = malloc(sizeof(S));
  return p;
}
```

Figure 2.7: Sample Code with conditional statements

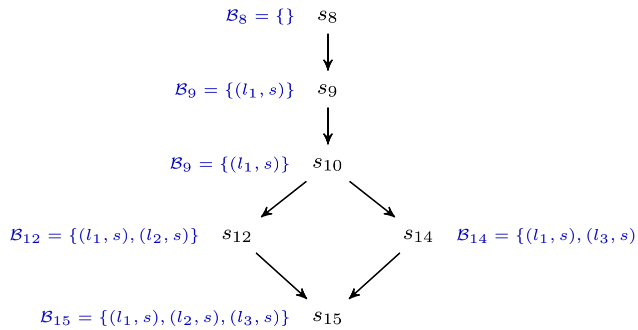

Figure 2.8: The blocks collected during the execution, s_i represents the state at line i and $\mathcal{B}_i$ represents the memory blocks collected at state s_i. $s = \textbf{sizeof}(\mathsf{S})$

that any de-allocated block must not overlap any other allocated block. We assume that we have enough memory space, and once a memory block is de-allocated, it cannot be reused. The predicate $\mathsf{non-overlap}$ is defined as a shortcut:

$$\mathsf{non\text{-}overlap}(l_1, s_1, l_2, s_2) \equiv l_1 +_{|\mathtt{ptr}|} s_1 \leq l_2 \vee l_2 +_{|\mathtt{ptr}|} s_2 \leq l_1$$

where (l_1, s_1) represents a memory block with base address l_1 and size s_1, and (l_2, s_2) is another block. This predicate specifies the non-overlapping of the two blocks.

One naive approach is to collect the memory blocks generated during the execution and apply $\mathsf{non-overlap}$ on each pair of them. Let $\mathcal{B}$ denote the collection of blocks. Then the disjointness can be expressed as

$$\forall (l_i, s_i), (l_j, s_j) \in \mathcal{B} \,.\, \bigwedge_{i \neq j} \mathsf{non\text{-}overlap}(l_i, s_i, l_j, s_j)$$

This approach, however, cannot support conditional statements.

Consider the sample code in Fig. 2.7. At the join point of "then" and "else" branches, the memory blocks $\mathcal{B}_{12}$ and $\mathcal{B}_{14}$ are merged into $\mathcal{B}_{15}$. Then, the disjointness constraint at program state s_{15} would be encoded as

$$\textsf{non-overlap}(l_1, s, l_2, s) \wedge \textsf{non-overlap}(l_1, s, l_3, s) \wedge \textsf{non-overlap}(l_2, s, l_3, s)$$

which is not correct. Because (l_2, s_2) and (l_3, s_3) cannot co-exist, it is incorrect to specify they are disjoint, and their disjointness with (l_1, s_1) must be guarded by the condition expression.

Sinz *et al.* proposed an alternative approach built on the *memory modification graph* in [38]. The memory modification graph is a transition graph over memory states. The vertices of the graph are memory states $m \in \mathbb{M}$, and a transition edge, denoted as (m, m'), represents the memory modification on the source state m which results in the target state m'. The modification includes operations like memory write, allocation and deallocation. Given a memory state m, the disjointness constraint is encoded as

$$\textsf{disjoint}(m) \equiv \bigwedge_{\substack{m_1 \preceq m_2 \preceq m \\ m_1 : \textsf{allocate}(l_1, s_1) \\ m_2 : \textsf{allocate}(l_2, s_2)}} (pc_{m_1} \wedge pc_{m_2} \implies \textsf{non-overlap}(l_1, s_1, l_2, s_2)) \quad (2.1)$$

where $m_1 \preceq m_2$ means there is a path from m_1 to m_2 in the graph; $m : \textsf{allocate}(l, s)$ denotes the memory block (l, s) is allocated at the memory state m; and pc_{m_1} and pc_{m_2} are the path conditions associated with memory states m_1 and m_2.

Fig. 2.9 is the memory modification graph built for the code in Fig. 2.7. Ac-

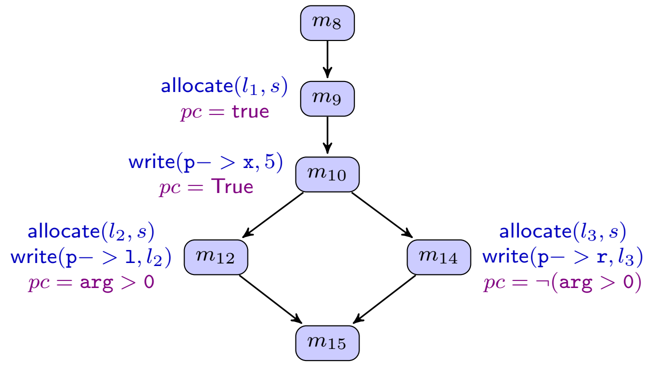

Figure 2.9: Memory modification graph

cording to (2.1), the disjointness constraint at memory state m_{15} is encoded as

$$\mathsf{disjoint}(m_{15}) \equiv \bigwedge \left(\begin{array}{c} \texttt{arg} > 0 \implies \mathsf{non\text{-}overlap}(l_1, s, l_2, s), \\ \neg\texttt{arg} > 0 \implies \mathsf{non\text{-}overlap}(l_1, s, l_3, s) \end{array} \right)$$

This approach is precise but rather inefficient. The memory modification graph must be maintained during the program execution. When encoding the disjointness constraint, we must search over the graph to find all the reachable states from the current memory state. Such an approach is known as *state-dependent*.

To address these issues, we propose a novel *state-independent* approach that can build the constraint automatically during the program execution without tracking the previously allocated blocks. To achieve it, we introduce

- predicate $\mathsf{disjoint}$, denoting that all the allocated blocks are disjoint;
- function $\mathsf{fun\text{-}disjoint} : \mathbb{V}_{\texttt{ptr}} \times \mathbb{N} \to \{\mathsf{True}, \mathsf{False}\}$, where $\mathsf{fun\text{-}disjoint}(x, y)$ denotes that memory block (x, y) is non-overlapping with previously allocated blocks.

They are components of the memory constraint ϕ. In a fresh memory constraint

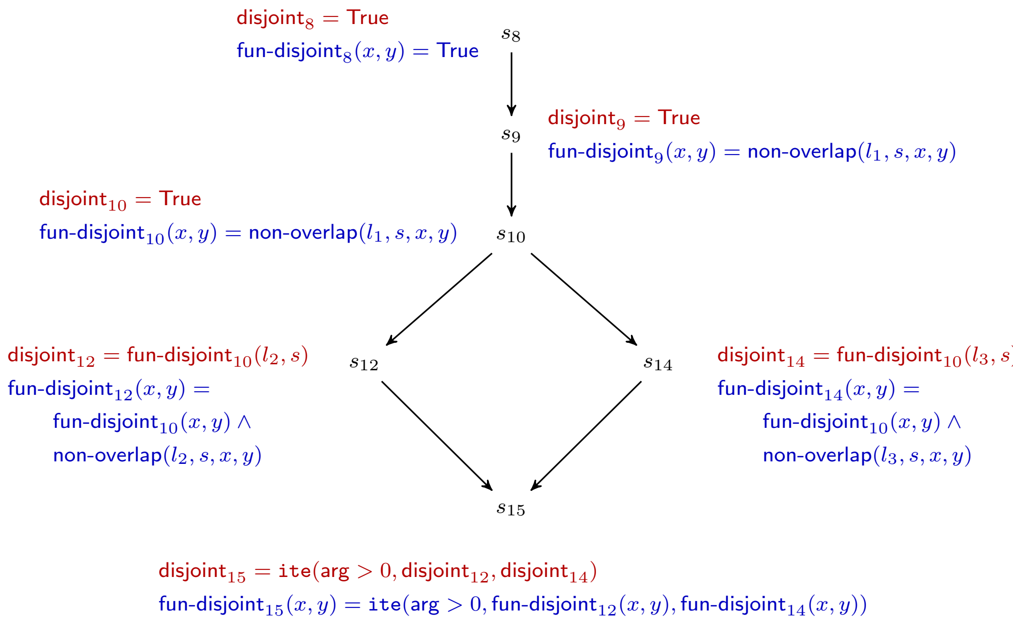

Figure 2.10: Disjointness constraints

ϕ_0, they are initialized as

$$\mathsf{disjoint}_0 = \mathsf{True}, \quad \mathsf{fun\text{-}disjoint}_0(x, y) = \mathsf{True} \tag{2.2}$$

When a new block (l, s) is allocated, they are updated as

$$\mathsf{disjoint}' = \mathsf{disjoint} \wedge \mathsf{fun\text{-}disjoint}(l, s) \tag{2.3}$$

$$\mathsf{fun\text{-}disjoint}'(x, y) = \mathsf{fun\text{-}disjoint}(x, y) \wedge \mathsf{non\text{-}overlap}(l, s, x, y) \tag{2.4}$$

When encoding $\phi_1 \sqcup_a \phi_2$ at a join point of branches, they are encoded as

$$\mathsf{disjoint}' = \mathtt{ite}(a, \mathsf{disjoint}_1, \mathsf{disjoint}_2) \tag{2.5}$$

$$\mathsf{fun\text{-}disjoint}'(x, y) = \mathtt{ite}(a, \mathsf{fun\text{-}disjoint}_1(x, y), \mathsf{fun\text{-}disjoint}_2(x, y)) \tag{2.6}$$

Looking again at the code in Fig. 2.7, the update of $\mathsf{disjoint}$ and $\mathsf{fun\text{-}disjoint}$ along the program execution is shown in Fig. 2.11. The disjointness constraint at the state s_{15} is encoded as

$$\mathsf{disjoint}_{15} \equiv \mathtt{ite}(\mathsf{arg} > 0, \mathsf{non\text{-}overlap}(l_1, s, l_2, s), \mathsf{non\text{-}overlap}(l_1, s, l_3, s))$$

2.4.2 Memory Safety

The constraints related to memory safety are: (1) valid memory access (i.e. the memory read/write operations affect only allocated memory); (2) valid frees (i.e. the pointer given as a parameter to a free instruction points to the base address of an allocated memory block that has not yet been de-allocated); and (3) no memory leaks (i.e. all allocated heap memory is de-allocated when the program ends).

Valid Memory Access. Let (x, y) denote a memory region to access where x is the memory address of dereference and y is the size of the access range. We say the region is valid to access, if there is an allocated (not yet de-allocated) memory block (l, s) containing (x, y). First, a predicate $\mathsf{contains}$ is defined as a shortcut

$$\mathsf{contains}(l, s, x, y) \equiv l \leq x < x +_{|\mathtt{ptr}|} y \leq l +_{|\mathtt{ptr}|} s$$

The encoding of a valid memory access constraint involves two elements:

- $size$, an array to track the size of allocated blocks, mapping from their base addresses to their sizes, where $\mathsf{read}(size, 0) = 0$;

- function $\mathsf{valid\text{-}deref} : \mathbb{V}_{\mathtt{ptr}} \times \mathbb{N} \rightarrow \{\mathsf{True}, \mathsf{False}\}$, where $\mathsf{valid\text{-}deref}(x, y)$ denotes $x, \ldots, x + y - 1$ is a valid sequence of addresses to access.

They are components of the memory constraint ϕ. In a fresh memory constraint ϕ_0, $size$ is a fresh array variable and

$$\mathsf{valid\text{-}deref}_0(x, y) = \mathsf{False} \tag{2.7}$$

When a new block (l, s) is allocated, they are updated as

$$\begin{aligned} \mathsf{valid\text{-}deref}'(x, y) &= \mathsf{valid\text{-}deref}(x, y) \lor (s \neq 0 \land \mathsf{contains}(l, s, x, y)) & (2.8) \\ size' &= \mathsf{write}(size, l, s) & (2.9) \end{aligned}$$

$\mathsf{valid\text{-}deref}'(x, y)$ ensures that any later access with address sequence $x, \ldots, x+y-1$ is *valid* if within the block (l, s).

When a block with base address l is de-allocated, they are updated as

$$\begin{aligned} \mathsf{valid\text{-}deref}'(x, y) &= \mathsf{valid\text{-}deref}(x, y) \land \neg\mathsf{contains}(l, \mathsf{read}(size, l), x, y) & (2.10) \\ size' &= \mathsf{write}(size, l, 0) & (2.11) \end{aligned}$$

Here, $\mathsf{valid\text{-}deref}'(x, y)$ ensures that any later access with address sequence $x, \ldots, x+y-1$ is *invalid* if within the de-allocated block.

When encoding $\phi_1 \sqcup_a \phi_2$ at a join point of branches, they are encoded as

$$\begin{aligned} \mathsf{valid\text{-}deref}'(x, y) &= \mathtt{ite}(a, \mathsf{valid\text{-}deref}_1(x, y), \mathsf{valid\text{-}deref}_2(x, y)) & (2.12) \\ size' &= \mathtt{ite}(a, size_1, size_2) & (2.13) \end{aligned}$$

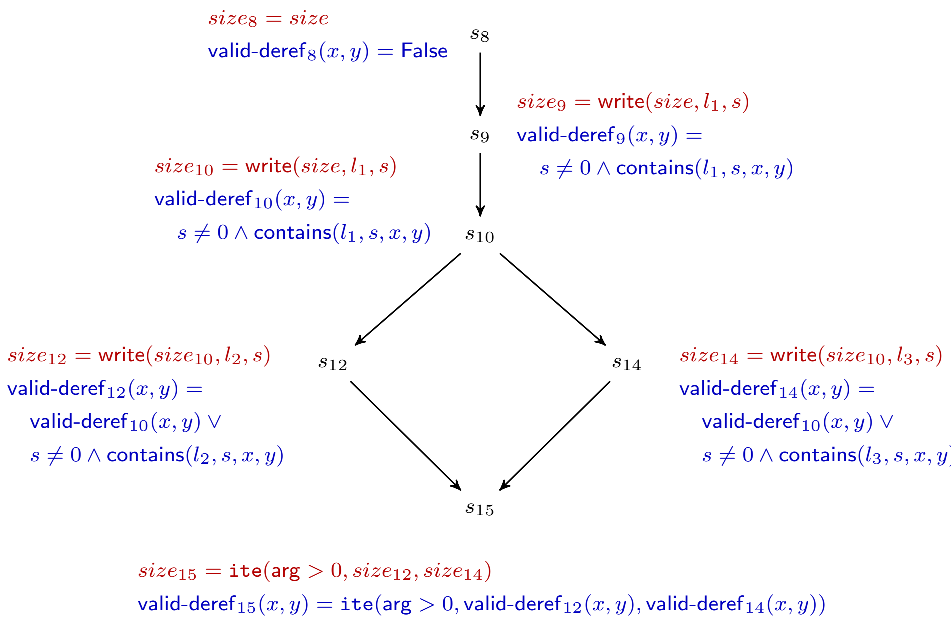

Figure 2.11: Valid memory access constraints

For the code in Fig. 2.7, the update of $\textsf{valid-deref}$ and $size$ along the program execution is shown in Fig. 2.11. The constraint of valid memory access at the state s_{15} is encoded as

$$\begin{aligned}\textsf{valid-deref}_{15}(x,y) \;&=\\ s \neq 0 \wedge &\left(\begin{array}{l}\textsf{contains}(l_1, s, x, y) \vee \\ \texttt{ite}(\textsf{arg} > 0, \textsf{contains}(l_2, s, x, y), \textsf{contains}(l_3, s, x, y))\end{array}\right)\end{aligned}$$

Valid-Free. The violation of valid-free includes invalid-free and double-free. Invalid-free happens if the address to be freed is not the base address of an allocated memory region. Double-free happens if the address to be freed is the base address of a memory region freed already.

In order to detect these bugs, an array $mark : \mathbb{V}_{\texttt{ptr}} \rightarrow \{\textsf{True}, \textsf{False}\}$ is introduced to track the state of memory blocks allocated on the heap, which is also a component of memory constraints ϕ. In the fresh memory constraints ϕ_0, $mark$ is a fresh array variable. It is updated when a new block is allocated on the heap and an address l is de-allocated, as follows:

$$mark' = \mathsf{write}(mark, l, \mathsf{True}), \quad (l, s) \text{ is allocated} \tag{2.14}$$

$$mark' = \mathsf{write}(mark, l, \mathsf{False}), \quad l \text{ is de-allocated} \tag{2.15}$$

When encoding $\phi_1 \sqcup_a \phi_2$ at a join point of branches, it is encoded as

$$mark' = \mathtt{ite}(a, mark_1, mark_2) \tag{2.16}$$

Then the constraint of valid free is encoded as

$$\mathsf{valid\text{-}free}(x) \equiv \mathsf{read}(mark, x) \tag{2.17}$$

For a free statement `free` e, where l is the result of evaluating e, it is double-free if $\mathsf{read}(mark, l) = \mathsf{False}$; if it is invalid-free, $\mathsf{read}(mark, l)$ can take any arbitrary value (either False or True). Thus we can check whether `free` e is valid by checking whether $\mathsf{read}(mark, l) = \mathsf{False}$ is unsatisfiable.

Valid-Memtrack. The check of memory leak can also depend on the array $mark$. At the beginning of the execution, $mark$ can be initialized as a constant array containing all False. At the end of the program, we could check if the final $mark$ still contains all False with quantifiers reasoning, however, this is notoriously difficult for most SMT solvers. To avoid quantifiers, we introduce a variable $\mathsf{memsize} \in \mathbb{N}$, denoting the total size of allocated memory blocks, which is a component of the memory constraint ϕ. In a fresh memory constraint ϕ_0, we have $\mathsf{memsize}_0 = 0$. It is updated when a block (l, s) is allocated on the heap or an address l is de-allocated, as following:

$$\mathsf{memsize}' = \mathsf{memsize} + s, \quad (l, s) \text{ is allocated} \tag{2.18}$$

$$\mathsf{memsize}' = \mathsf{memsize} - \mathsf{read}(size, l), \quad l \text{ is de-allocated} \tag{2.19}$$

Note that $\mathsf{memsize}$ can also be used to check whether a given memory limit is exceeded.

When encoding $\phi_1 \sqcup_a \phi_2$ at a join point of branches, it is encoded as

$$\mathsf{memsize}' = \mathtt{ite}(a, \mathsf{memsize}_1, \mathsf{memsize}_2) \tag{2.20}$$

At the end of the execution, the constraint of no memory leak is encoded as

$$\mathsf{no\text{-}memory\text{-}leak} \equiv \mathsf{memsize} = 0 \tag{2.21}$$

There is a memory leak if $\mathsf{memsize}$ is positive. On the other hand, if $\mathsf{memsize}$ is negative, then there is an invalid-free, since $\mathsf{memsize} - \mathsf{read}(size, l)$ can take an any arbitrary value considering $\mathsf{read}(size, l)$ could be anything if l is not the base address of any allocated memory blocks.

2.4.3 Memory Constraints

Based on the encoding of memory constraints, we now are able to fill in the missing parts of statement semantics: a memory constraint ϕ is a tuple

$$\phi = \langle \mathsf{disjoint}, \mathsf{fun\text{-}disjoint}, \mathsf{valid\text{-}deref}, size, mark, \mathsf{memsize} \rangle$$

The initial value of ϕ is

$$\langle \mathsf{True}, \lambda x, y.\mathsf{True}, \lambda x, y.\mathsf{False}, size, mark, 0 \rangle$$

where $size$ and $mark$ are fresh array variables. In the following, we use the judgement $\langle \rho, s \rangle \Downarrow \rho'$ to denote the transition of the state of memory partitions tracked in ρ. Given a memory partition identifier k, let $\rho(k) = \langle m_k, \phi_k \rangle$ where ϕ_k is the memory constraint associated with the memory partition. We use the notation $\phi_k[x \mapsto y]$ to specify that the constraint component x is updated to y.

▶ **Variable Declaration**

$$\langle \rho, \texttt{declare v} \rangle \Downarrow \rho'$$

Let $k_v = \mathsf{partition}(v)$ and $\forall k \in \mathsf{partitions}(k_v)$

- if $k \in \texttt{dom}(\rho)$ and $\rho(k) = \langle m_k, \phi_k \rangle$, then $\rho'(k) = \langle m_k, \phi'_k \rangle$, where

$$\begin{aligned}
\phi'_k \;&=\; \phi_k \begin{bmatrix} \mathsf{disjoint} & \mapsto & \mathsf{disjoint}_k \wedge \mathsf{fun\text{-}disjoint}_k(\epsilon(v), \mathsf{sizeof}(v)) \\ \mathsf{fun\text{-}disjoint} & \mapsto & \mathsf{fun\text{-}disjoint}'_k \\ \mathsf{valid\text{-}deref} & \mapsto & \mathsf{valid\text{-}deref}'_k \end{bmatrix} \\
\mathsf{fun\text{-}disjoint}'_k(x, y) \;&=\; \mathsf{fun\text{-}disjoint}_k(x, y) \wedge \mathsf{non\text{-}overlap}(\epsilon(v), \mathsf{sizeof}(v), x, y) \\
\mathsf{valid\text{-}deref}'_k(x, y) \;&=\; \mathsf{valid\text{-}deref}_k(x, y) \vee \mathsf{contains}(\epsilon(v), \mathsf{sizeof}(v), x, y)
\end{aligned}$$

- otherwise $\rho'(k) = \langle m_k, \phi_k \rangle$ where

$$\begin{aligned}
\phi_k &= \langle \mathsf{True}, \mathsf{fun\text{-}disjoint}_k, \mathsf{valid\text{-}deref}_k, size_k, mark_k, 0 \rangle \\
\mathsf{fun\text{-}disjoint}_k(x, y) &= \mathsf{non\text{-}overlap}(\epsilon(v), \mathsf{sizeof}(v), x, y) \\
\mathsf{valid\text{-}deref}_k(x, y) &= \mathsf{contains}(\epsilon(v), \mathsf{sizeof}(v), x, y)
\end{aligned}$$

with $size_k$ and $mark_k$ fresh array variables.

▶ **Memory Allocation**

$$\langle \rho, e_1 = \mathtt{malloc}(e_2) \rangle \Downarrow \rho'$$

Let $k = \mathsf{partition}(e_1)$ and $k_* = \mathsf{ptsto}(k)$, the memory partition identifier of the newly allocated memory region with base address region and size a (the evaluation result of e_2). Then $\forall k' \in \mathsf{partitions}(k_*)$,

- if $k' \in \mathtt{dom}(\rho)$ and $\rho(k') = \langle m_{k'}, \phi_{k'} \rangle$, then $\rho'(k') = \langle m_{k'}, \phi'_{k'} \rangle$, in which the memory constraint is updated to $\phi'_{k'}$ where

$$\begin{aligned}
\phi'_{k'} &= \phi_{k'} \left[\begin{array}{rcl}
\mathsf{disjoint} & \mapsto & \mathsf{disjoint}_{k'} \wedge \mathsf{fun\text{-}disjoint}_{k'}(\mathsf{region}, a) \\
\mathsf{fun\text{-}disjoint} & \mapsto & \mathsf{fun\text{-}disjoint}'_{k'} \\
\mathsf{valid\text{-}deref} & \mapsto & \mathsf{valid\text{-}deref}'_{k'} \\
size & \mapsto & \mathsf{write}(size_{k'}, \mathsf{region}, a) \\
mark & \mapsto & \mathsf{write}(mark_{k'}, \mathsf{region}, \mathsf{True}) \\
\mathsf{memsize} & \mapsto & \mathsf{memsize}_{k'} + a
\end{array} \right] \\
\mathsf{fun\text{-}disjoint}'_{k'}(x, y) &= \mathsf{fun\text{-}disjoint}_{k'}(x, y) \wedge \mathsf{non\text{-}overlap}(\mathsf{region}, a, x, y) \\
\mathsf{valid\text{-}deref}'_{k'}(x, y) &= \mathsf{valid\text{-}deref}_{k'}(x, y) \vee (a \neq 0 \wedge \mathsf{contains}(\mathsf{region}, a, x, y))
\end{aligned}$$

- otherwise, $\rho'(k') = \langle m_{k'}, \phi_{k'} \rangle$,

$$\begin{array}{rcl} \phi_{k'} & = & \left\langle \begin{array}{c} \mathsf{True}, \mathsf{fun\text{-}disjoint}_{k'}, \mathsf{valid\text{-}deref}_{k'}, \\ \mathsf{write}(size_{k'}, \mathsf{region}, a), \mathsf{write}(mark_{k'}, \mathsf{region}, \mathsf{True}), a \end{array} \right\rangle \\ \mathsf{fun\text{-}disjoint}_{k'}(x, y) & = & \mathsf{non\text{-}overlap}(\mathsf{region}, a, x, y) \\ \mathsf{valid\text{-}deref}_{k'}(x, y) & = & a \neq 0 \wedge \mathsf{contains}(\mathsf{region}, a, x, y) \end{array}$$

where $size_{k'}$ and $mark_{k'}$ are fresh array variables.

▶ **Memory Deallocation**

$$\langle \rho, \texttt{free } e \rangle \Downarrow \rho'$$

Let $k = \mathsf{ptsto}(\mathsf{partition}(e))$, the memory partition identifier of the memory region pointed by e. Then $\forall k' \in \mathsf{partitions}(k)$, $\rho(k') = \langle m_{k'}, \phi_{k'} \rangle$ and $\rho'(k') = \langle m_{k'}, \phi'_{k'} \rangle$, in which the memory constraint is updated to $\phi'_{k'}$ where

$$\begin{array}{rcl} \phi'_{k'} & = & \phi_{k'} \left[\begin{array}{rcl} \mathsf{valid\text{-}deref} & \mapsto & \mathsf{valid\text{-}deref}'_{k'} \\ size & \mapsto & \mathsf{write}(size_{k'}, \mathsf{region}, 0) \\ mark & \mapsto & \mathsf{write}(mark_{k'}, \mathsf{region}, \mathsf{False}) \\ \mathsf{memsize} & \mapsto & \mathsf{memsize}_{k'} - \mathsf{read}(size_{k'}, \mathsf{region}) \end{array} \right] \\ \mathsf{valid\text{-}deref}'_{k'}(x, y) & = & \mathsf{valid\text{-}deref}_{k'}(x, y) \wedge \neg \mathsf{contains}(\mathsf{region}, \mathsf{read}(size_{k'}, \mathsf{region}), x, y) \end{array}$$

▶ **Assertion**

$$\langle \rho, \texttt{assert } e \rangle \Downarrow \rho$$

The disjointness constraint $\mathsf{disjoint}(\rho)$ is defined as

$$\mathsf{disjoint}(\rho) \equiv \bigwedge_{k \in \mathtt{dom}(\rho)} \mathsf{disjoint}_k \tag{2.22}$$

▶ **Conditional**

$$\langle \rho, \texttt{if}(e)\ s_1\ \texttt{else}\ s_2 \rangle \Downarrow \rho'$$

The join of memory constraints $\phi_{k_1} \sqcup_a \phi_{k_2}$ (a is the evaluation result of conditional expression e), is defined as

$$\phi_{k_1} \sqcup_a \phi_{k_2} \quad \equiv \quad \begin{bmatrix} \textsf{disjoint} & \mapsto & \texttt{ite}(a, \textsf{disjoint}_{k_1}, \textsf{disjoint}_{k_2}) \\ \textsf{fun-disjoint}(x, y) & \mapsto & \texttt{ite}(a, \textsf{fun-disjoint}_{k_1}(x, y), \textsf{fun-disjoint}_{k_2}(x, y) \\ \textsf{valid-deref}(x, y) & \mapsto & \texttt{ite}(a, \textsf{valid-deref}_{k_1}(x, y), \textsf{valid-deref}_{k_2}(x, y) \\ size & \mapsto & \texttt{ite}(a, size_{k_1}, size_{k_2}) \\ mark & \mapsto & \texttt{ite}(a, mark_{k_1}, mark_{k_2}) \\ \textsf{memsize} & \mapsto & \texttt{ite}(a, \textsf{memsize}_{k_1}, \textsf{memsize}_{k_2}) \end{bmatrix} \tag{2.23}$$

Chapter 3

Cell-based Points-to Analysis

In this chapter, we present the cell-based points-to analysis, which is an extension of Steensgaard's field-sensitive points-to analysis. It yields more precise results for arrays of records and heap allocated records. Section 3.1 introduces the analysis with motivating examples. Section 3.2 gives a formal presentation. Section 3.3 provides the soundness proof.

3.1 Overview

While Steensgaard's field-sensitive algorithm does improve the precision of points-to analysis when dealing with records that are allocated statically, it cannot do the same for dynamically allocated records. Furthermore, it always collapses arrays into a single alias group, meaning that the points-to analysis cannot distinguish fields that occur inside array elements. To address these issues, we developed a novel *cell-based* points-to analysis, which is *fully* field-sensitive. It can precisely capture field overlapping (field aliasing) induced by union types, pointer casts and pointer arithmetic. The analysis is built on the application binary interface (ABI).

In this section, we describe this analysis at a high level and give several examples.

A *cell*[1] is a generalization of an alias group. Initially, each program expression that corresponds to a memory location at runtime (i.e. an l-value) is associated with a unique cell whose *size* is a positive integer denoting the size (in bytes) of the values that expression can have. In addition, each cell has a type, which is *scalar* unless its associated program expression is of record or union type (in which case the cell type is *record* or *union* respectively). Under certain conditions, the analysis may merge cells. If two cells of two different sizes are merged, then the result is a cell whose size is $\top$. The graph maintains an invariant that the locations associated with any two scalar cells are *always disjoint*, which makes the memory partition using the graph possible.

Our analysis creates a points-to graph whose vertices are cells. The graph has two kinds of edges. A *points-to* edge $\alpha \rightharpoonup \beta$ denotes that dereferencing some expression associated with cell α yields an address that must be in one of the locations associated with cell β. Unlike traditional field-sensitive analyses, inner cells may be nested in more than one outer cell. Thus, we use additional graph edges to represent containment relations. A *contains edge* $\alpha \hookrightarrow_{i,j} \beta$ denotes that cell α is of record type and that β is associated with a field of the record whose location is at an offset of i from the record start and whose size in bytes is $j - i$.

Fig. 3.1 shows a simple example. On the left is the memory layout of a singly-linked list with one element. The element is a record with two fields, a data value and a *next* pointer (which points back to the element in this case). The graph, shown on the right, contains three cells. The square cell is associated with the entire record element and the round cells with the inner fields (here and in

[1]We borrowed this term from Mine [32].

Figure 3.1: Concrete memory state and its graph representation

the other points-to graphs below, we follow the convention that square cells are of record or union type and round cells are of scalar type). The dashed edge is a points-to edge from the next field to the record cell, and the solid edges are contains edges from the record cell to the field cells. These edges are labeled with their corresponding starting and ending offsets within the record.

Arrays. Arrays are handled by merging all of the cells associated with the array's elements into a single cell. Note that if the array elements are records, then this preserves the cell type and size, allowing arrays of records to be treated more precisely, with a separate inner cell for each record field. Thus, identical fields in different array elements do share the same cell, but different fields in any two elements will be assigned different cells. This is a key innovation as even when the array size is unknown, the cell size is known. For example, suppose an array is dynamically allocated as `uint32_t* p = (uint32_t *) malloc(sizeof(uint32_t) * m)`.[2] The data size of the allocated array is not known statically, but each element is 4 bytes long. Knowing this, we can model the cell using a memory array whose elements are 4 bytes long. This further improves the precision and performance of the

[2]Function `malloc` returns a `void` pointer, and such a pointer must be cast to a non-`void` pointer for further access to the allocated region. The cell size of the region is initialized to the byte-size of the non-`void` type.

```
typedef struct SList1 {
  struct SList1 *next;
  uint32 data1;
} SList1;
typedef struct SList2 {
  struct SList2 *next;
  uint16 data2;
} SList2;
typedef union SList {
  union SList* next;
  SList1 sl1;
  SList2 sl2;
} SList;
SList l;
```

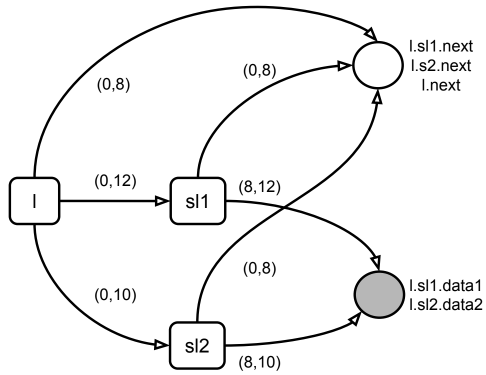

Figure 3.2: Points-to graph with union type

```
struct {
  int32 c;
  struct {int32 a; int64 b;} t;
} s;
...
*(&s.t.a + i) = 0;
```

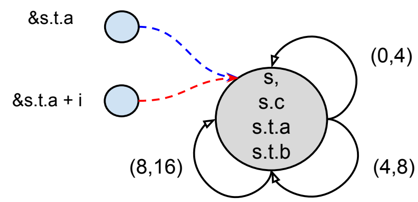

Figure 3.3: Points-to graph with pointer arithmetic

memory model.

Union Types. Consider the code in Fig. 3.2. The union type `SList` has a pointer field and two record fields, each of which offers a different view of the same memory region. The corresponding graph representation is shown on the right. Contains edges link the union cell with cells for each of its fields. Note that a single cell represents the expressions `l.next`, `l.sl1.next` and `l.sl2.next`. This is because we can determine based on the contains edges that these all refer to the same memory location. Another cell (in gray) represents both `l.sl1.data1` and `l.sl2.data2` for the same reason. In this case, however, because these fields are of different sizes, the cell size is $\top$.

```
typedef struct DList {
  struct DList* left;
  struct DList* right;
} DList;
typedef struct SList {
  DList dl;
  struct SList* next;
} SList;
SList l;
((DList*) &l.dl.right)->right = NULL;
```

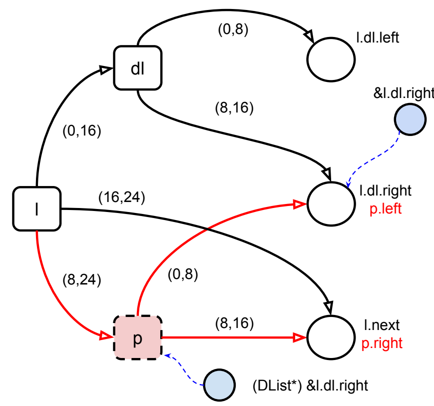

Figure 3.4: Points-to graph with pointer cast

Pointer Arithmetic. Consider next the code in Fig. 3.3. Any field in the record `t` (and even in the outer record `s`) can be modified by the assignment `&s.t.a + i`, since the value of `i` is unknown. In this case, we merge all of the record (and inner field) cells into a single scalar cell. The resulting points-to graph is shown on the right.

Pointer Casts. Casting creates an alternative view of a memory region. In this sense, it is similar to a union. To model this, a fresh cell is added to the points-to graph representing the new view. Consider the code in Fig. 3.4. The field `l.dl.right` is cast to be of type `DList`. As shown in the graph on the right, a new cell `p` is created whose inner cells overlap with the original fields `l.dl.right` and `l.next`.

3.2 Constraint-based Analysis

In this section, we formalize the cell-based field-sensitive points-to analysis described above using a constraint framework. Our constraint-based program analysis is divided into two phases: constraint generation and constraint resolution. The

t	::=	uint8 | int8 | ... | int64	integer types
	|	ptr	pointer types
	|	struct{ $\mathtt{t}_1\mathtt{f}_1; \ldots \mathtt{t}_n\mathtt{f}_n;$ } S	record types
	|	union{ $\mathtt{t}_1\mathtt{f}_1; \ldots \mathtt{t}_n\mathtt{f}_n;$ } U	union types
op_b	::=	$+ \mid - \mid * \mid /$	operators

e	::=	n	constant
	|	x	variable
	|	$*\,e$	dereference
	|	$\&e$	address of
	|	(t*) e	cast
	|	e.f	field selection
	|	(t*) malloc(e)	heap allocation
	|	$e_1\ op_b\ e_2$	binary operation
	|	$e_1 =_{\mathtt{t}} e_2$	assignment, t is scalar
	|	e_1, e_2	sequencing

Figure 3.5: Language syntax

constraint generation phase produces constraints from the program source code in a syntax-directed manner. The constraint resolution phase then computes a solution of the constraints in the form of a cell-based field-sensitive points-to graph. The resulting graph describes a safe partitioning of the memory for all reachable states of the program.

3.2.1 Language and Constraints

For the formal treatment of our analysis, we consider the idealized C-like language shown in Fig. 3.5. To simplify the presentation, complex assignments are broken down to simpler assignments between expressions of scalar types, static arrays are represented as pointers to dynamically allocated regions, and a single type `ptr` is used to represent all pointer types. Function definitions, function calls, and function pointers are omitted. [3]

Let $\mathbb{C}$ be an infinite set of *cell variables* (denoted τ or τ_i). We will use cell variables to assign program expressions to cells in the resulting points-to graph. To do so, we assume that each subexpression e' occurring in an expression e is labeled with a *unique* cell variable τ, with the exception that program variables

[3]They can be handled using a straightforward adaptation of Steensgaard's approach.

`x` are always assigned the same cell variable, τ_x. Cell variables associated with program variables are called *source* variables. To avoid notational clutter, we do not make cell variables explicit in our grammar definition. Instead, we write $e : \tau$ to indicate that the expression e is labeled by τ.

3.2.1.1 Constraints.

The syntax of our constraint language is defined as follows:

$$
\begin{array}{lcll}
\eta & ::= & i \mid \top \mid \mathsf{size}(\tau) & \quad i \in \mathbb{N} \\
\phi & ::= & i < \eta \mid \eta_1 = \eta_2 \mid \tau_1 = \tau_2 \mid \tau_1 \rightharpoonup \tau_2 \mid \tau_1 \hookrightarrow_{i,j} \tau_2 \mid \tau_1 \trianglelefteq \tau_2 & \\
 & \mid & \mathsf{source}(\tau) \mid \mathsf{scalar}(\tau) \mid \mathsf{cast}(i, \tau_1, \tau_2) \mid \mathsf{collapsed}(\tau) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 &
\end{array}
$$

Here, a term η denotes a *cell size*. The constant $\top$ indicates an unknown cell size. A constraint ϕ is a positive Boolean combination of cell size constraints, equalities on cell variables, points-to edges $\tau_1 \rightharpoonup \tau_2$, contains edges $\tau_1 \hookrightarrow_{i,j} \tau_2$ and special predicates whose semantics we describe in detail below. We additionally introduce syntactic short-hands for certain constraints. Namely, we write $i \sqsubseteq \eta$ to stand for the constraint $i = \eta \vee \eta = \top$, $i \leq \eta$ to stand for $i < \eta \vee i = \eta$, and $i \preceq \eta$ to stand for $i \leq \eta \vee \eta = \top$.

Constraints are interpreted in cell-based field-sensitive points-to graphs (CFPGs). A CFPG is a tuple $G = (C, \mathit{cell}, \mathit{size}, \mathit{source}, \mathit{scalar}, \mathit{contains}, \mathit{ptsto})$ where

- C is a finite set of cells,
- $\mathit{cell} : \mathbb{C} \to C$ is an assignment from cell variables to cells,
- $\mathit{size} : C \to \mathbb{N} \cup \{\top\}$ is an assignment from cells to cell sizes,

- *source* $\subseteq C$ is a set of *source cells*,

- *scalar* $\subseteq C$ is a set of *scalar cells*,

- *contains* $\subseteq C \times \mathbb{N} \times \mathbb{N} \times C$ is a *containment relation* on cells, and

- *ptsto* $: C \to C$ is a *points-to map* on cells.

For $c_1, c_2 \in C$, and $i, j \in \mathbb{N}$, we write $c_1 \overset{G}{\hookrightarrow}_{i,j} c_2$ as notational sugar for $(c_1, i, j, c_2) \in$ *contains*, and similarly $c_1 \overset{G}{\rightharpoonup} c_2$ for $\mathit{ptsto}(c_1) = c_2$. Let *contains*′ be the projection of *contains* onto $C \times C$: $\mathit{contains}'(c_1, c_2) \equiv \exists\, i, j.\ \mathit{contains}(c_1, i, j, c_2)$.

The functions and relations of G must satisfy the following consistency properties. These properties formalize the intuition of the containment relation and the roles played by source and scalar cells:

- the *contains*′ relation is reflexive (for cells of known size).

$$\forall\, c \in C.\ \mathit{size}(c) \neq \top \implies c \overset{G}{\hookrightarrow}_{0,\mathit{size}(c)} c \tag{3.1}$$

- the *contains*′ relation is transitive.

$$\forall \begin{pmatrix} c_1, c_2, c_3 \in C, \\ i_1, i_2, j_1, j_2 \in \mathbb{N} \end{pmatrix} .\ c_1 \overset{G}{\hookrightarrow}_{i_1,j_1} c_2 \wedge c_2 \overset{G}{\hookrightarrow}_{i_2,j_2} c_3 \implies c_1 \overset{G}{\hookrightarrow}_{i_1+i_2,i_1+j_2} c_3 \tag{3.2}$$

- the *contains*′ relation is anti-symmetric.

$$\forall \left(c_1, c_2 \in C i_1, i_2, j_1, j_2 \in \mathbb{N} \right) .\ c_1 \overset{G}{\hookrightarrow}_{i_1,j_1} c_2 \wedge c_1 \overset{G}{\hookrightarrow}_{i_2,j_2} c_2 \implies c_1 = c_2 \tag{3.3}$$

- the *contains* relation must satisfy the following linearity property:

$$\forall \begin{pmatrix} c_1, c_2, c_3 \in C, \\ i_1, i_2, j_1, j_2 \in \mathbb{N} \end{pmatrix} . \begin{pmatrix} i_1 \leq i_2 < j_2 \leq j_1 \wedge \\ c_1 \overset{G}{\hookrightarrow}_{i_1,j_1} c_2 \wedge \\ c_1 \overset{G}{\hookrightarrow}_{i_2,j_2} c_3 \end{pmatrix} \implies c_2 \overset{G}{\hookrightarrow}_{i_2-i_1,j_2-i_1} c_3 \tag{3.4}$$

- cells that are of unknown size or that point to other cells must be scalar:

$$\forall\, c \in C.\ size(c) = \top \implies c \in scalar \tag{3.5}$$

$$\forall\, c, c' \in C.\ c \overset{G}{\rightharpoonup} c' \implies c \in scalar \tag{3.6}$$

- source cells are not contained in and scalar cells do not contain other cells:

$$\forall\, c, c' \in C, i, j \in \mathbb{N}.\ c \overset{G}{\hookrightarrow}_{i,j} c' \wedge c' \in source \implies c = c' \tag{3.7}$$

$$\forall\, c, c' \in C, i, j \in \mathbb{N}.\ c \in scalar \wedge c \overset{G}{\hookrightarrow}_{i,j} c' \implies c = c' \tag{3.8}$$

- cell sizes must be consistent with the *contains* relation:

$$\forall \begin{pmatrix} c_1, c_2 \in C, \\ i, j \in \mathbb{N} \end{pmatrix} . c_1 \overset{G}{\hookrightarrow}_{i,j} c_2 \implies \begin{pmatrix} 0 \leq i < j \wedge \\ j \preceq size(c_1) \wedge j - i \sqsubseteq size(c_2) \end{pmatrix} \tag{3.9}$$

- two scalar cells must be equivalent if they are overlapped. First, We first express the notion of overlap $overlap^G(c_1, c_2)$ formally:

$$\exists \begin{pmatrix} c \in C, \\ i_1, i_2, j_1, j_2 \in \mathbb{N} \end{pmatrix} . \begin{pmatrix} c \overset{G}{\hookrightarrow}_{i_1,j_1} c_1 \wedge \\ c \overset{G}{\hookrightarrow}_{i_2,j_2} c_2 \end{pmatrix} \wedge \begin{pmatrix} i_1 \leq i_2 < j_1 \vee \\ i_2 \leq i_1 < j_2 \end{pmatrix} \tag{3.10}$$

Then

$$\forall c_1, c_2 \in C.\, overlap^G(c_1, c_2) \wedge c_1 \in scalar \wedge c_2 \in scalar \implies c_1 = c_2 \quad (3.11)$$

Semantics of Constraints. Let G be a CFPG with components as above. For a cell variable $\tau \in \mathbb{C}$, we define $\tau^G = cell(\tau)$ and for a size term η we define $\eta^G = size(\tau^G)$ if $\eta = \mathsf{size}(\tau)$ and $\eta^G = \eta$ otherwise. The semantics of a constraint ϕ is given by a satisfaction relation $G \models \phi$, which is defined recursively on the structure of ϕ in the expected way. Size and equality constraints are interpreted in the obvious way using the term interpretation defined above. Though, note that we define $G \not\models i < \top$ and $G \not\models i = \top$.

Points-to constraints $\tau_1 \rightharpoonup \tau_2$ are interpreted by the points-to map $\tau_1^G \overset{G}{\rightharpoonup} \tau_2^G$; contains constraints $\tau_1 \hookrightarrow_{i,j} \tau_2$ are interpreted by the containment relation $\tau_1^G \overset{G}{\hookrightarrow}_{i,j} \tau_2^G$; and source and scalar are similarly interpreted by *source* and *scalar*.

Intuitively, a cast predicate $\mathsf{cast}(k, \tau_1, \tau_2)$ states that cell τ_2 is of size k and is obtained by a pointer cast from cell τ_1. Thus, any source cell that contains τ_1 at offset i must also contain τ_2 at that offset. That is, $G \models \mathsf{cast}(k, \tau_1, \tau_2)$ iff:

$$\forall c \in C, i, j \in \mathbb{N}.\, c \in source \wedge c \overset{G}{\hookrightarrow}_{i,j} \tau_1^G \implies c \overset{G}{\hookrightarrow}_{i,i+k} \tau_2^G \quad (3.12)$$

The predicate $\mathsf{collapsed}(\tau)$ indicates that τ points to a cell c that may be accessed in a type-unsafe manner, e.g., due to pointer arithmetic. All cells that contain a cell overlapping c should be collapsed. Then, $G \models \mathsf{collapsed}(\tau)$ iff

$$\forall c, c_1, c_2 \in C, i, j \in \mathbb{N}.\, \tau^G \overset{G}{\rightharpoonup} c \wedge c_1 \overset{G}{\hookrightarrow}_{i,j} c_2 \wedge overlap^G(c, c_2) \implies c = c_1 \quad (3.13)$$

The predicate $\tau_1 \trianglelefteq \tau_2$ (taken from [39]) is used to state the equivalence of the points-to content of τ_1 and τ_2. Formally, $G \models \tau_1 \trianglelefteq \tau_2$ iff

$$\forall c \in C.\, \tau_1^G \overset{G}{\rightharpoonup} c \implies \tau_2^G \overset{G}{\rightharpoonup} c \tag{3.14}$$

3.2.2 Constraint Generation

The first phase of our analysis generates constraints from the target program in a syntax-directed bottom-up fashion. The constraint generation is described by the inference rules in Fig. 3.6. Recall that each program expression e is labeled with a cell variable τ. The judgment form $e : \tau | \phi$ means that for the expression e labeled by the cell variable τ, we infer the constraint ϕ over the cell variables of e (including τ).

For simplicity, we assume the target program is well-typed. Our analysis relies on the type system to infer the byte-sizes of expressions and the field-layout within records and unions. To this end, we assume a type environment $\mathcal{T}$ that assigns C types to program variables. Moreover, we assume the following functions: $\mathit{typeof}(\mathcal{T}, e)$ infers the type of an expression following the standard type inference rules in the C language; $|\mathtt{t}|$ returns the byte-size of the type $\mathtt{t}$; and $\mathit{offset}(\mathtt{t}, \mathtt{f})$ returns the offset of a field $\mathtt{f}$ from the beginning of its enclosing record type $\mathtt{t}$. Finally, $\mathit{isScalar}(\mathtt{t})$ returns true iff the type $\mathtt{t}$ is an integer or pointer type.

The inference rules are inspired by the formulation of Steensgaard's field-insensitive analysis due to Forster and Aiken [18]. We adapt them to our cell-based field-sensitive analysis. Note that implications of the form $\mathit{isScalar}(\mathtt{t}) \implies \mathsf{scalar}(\tau)$, which we use in some of the rules, are directly resolved during the rule application and do not yield disjunctions in the generated constraints.

$$\textsc{Const}\ \frac{\boxed{\tau = \tau}^{\phi}}{n : \tau \mid \phi} \qquad \textsc{Seq}\ \frac{\begin{array}{c} e_1 : \tau_1 \mid \phi_1 \quad e_2 : \tau_2 \mid \phi_2 \\ \boxed{\tau = \tau_2}^{\phi} \end{array}}{e_1, e_2 : \tau \mid \phi_1 \wedge \phi_2 \wedge \phi}$$

$$\textsc{Assg}\ \frac{\begin{array}{c} e_1 : \tau_1 \mid \phi_1 \quad e_2 : \tau_2 \mid \phi_2 \\ \boxed{\tau_2 \trianglelefteq \tau_1 \quad \tau = \tau_2}^{\phi} \end{array}}{e_1 = e_2 : \tau \mid \phi_1 \wedge \phi_2 \wedge \phi} \qquad \textsc{Var}\ \frac{\boxed{\begin{array}{c} \mathtt{t} = \mathit{typeof}(\mathcal{T}, \mathtt{x}) \quad \mathsf{source}(\tau) \\ \mathit{isScalar}(\mathtt{t}) \implies \mathsf{scalar}(\tau) \\ \tau \hookrightarrow_{0,|\mathtt{t}|} \tau \end{array}}^{\phi}}{\mathtt{x} : \tau \mid \phi}$$

$$\textsc{Addr}\ \frac{\begin{array}{c} e : \tau \mid \phi_1 \\ \boxed{\begin{array}{c} \mathsf{source}(\tau') \\ \tau' \rightharpoonup \tau \quad \tau' \hookrightarrow_{0,|\mathtt{ptr}|} \tau' \end{array}}^{\phi} \end{array}}{\&e : \tau' \mid \phi_1 \wedge \phi} \qquad \textsc{Deref}\ \frac{\begin{array}{c} e : \tau \mid \phi_1 \\ \boxed{\begin{array}{c} \mathtt{t} = \mathit{typeof}(\mathcal{T}, *e) \\ \mathit{isScalar}(\mathtt{t}) \implies \mathsf{scalar}(\tau') \\ \tau \rightharpoonup \tau' \quad \tau' \hookrightarrow_{0,|\mathtt{t}|} \tau' \end{array}}^{\phi} \end{array}}{*e : \tau' \mid \phi_1 \wedge \phi}$$

$$\textsc{Malloc}\ \frac{\begin{array}{c} \mathtt{malloc}_l : \tau \\ \boxed{\begin{array}{c} \mathit{isScalar}(\mathtt{t}) \implies \mathsf{scalar}(\tau) \\ \mathsf{source}(\tau) \quad \mathsf{source}(\tau') \\ \tau' \rightharpoonup \tau \quad \tau' \hookrightarrow_{0,|\mathtt{ptr}|} \tau' \quad \tau \hookrightarrow_{0,|\mathtt{t}|} \tau \end{array}}^{\phi} \end{array}}{(\mathtt{t}*)\mathtt{malloc}_l(e) : \tau' \mid \phi}$$

$$\textsc{Dir-Sel}\ \frac{\begin{array}{c} e : \tau \mid \phi_1 \\ \boxed{\begin{array}{c} \mathtt{t} = \mathit{typeof}(\mathcal{T}, e) \\ o = \mathit{offset}(\mathtt{t}, \mathtt{f}) \\ \mathit{isScalar}(\mathtt{t.f}) \implies \mathsf{scalar}(\tau_f) \\ \tau \hookrightarrow_{o,o+|\mathtt{t.f}|} \tau_f \quad \tau_f \hookrightarrow_{0,|\mathtt{t.f}|} \tau_f \end{array}}^{\phi} \end{array}}{e.\mathtt{f} : \tau_f \mid \phi_1 \wedge \phi} \qquad \textsc{Cast}\ \frac{\begin{array}{c} e : \tau_1 \mid \phi_1 \qquad (\mathtt{t}*)_l : \tau_2' \\ \boxed{\begin{array}{c} \mathit{isScalar}(\mathtt{t}) \implies \mathsf{scalar}(\tau_2') \\ \tau_1 \rightharpoonup \tau_1' \quad \tau_2 \rightharpoonup \tau_2' \\ \tau_2 \hookrightarrow_{0,|\mathtt{ptr}|} \tau_2 \quad \tau_2' \hookrightarrow_{0,|\mathtt{t}|} \tau_2' \\ \mathsf{cast}(|\mathsf{t}|, \tau_1', \tau_2') \end{array}}^{\phi} \end{array}}{(\mathtt{t}*)_l e : \tau_2 \mid \phi_1 \wedge \phi}$$

$$\textsc{Arith-Op}\ \frac{\begin{array}{c} e_1 : \tau_1 \mid \phi_1 \qquad e_2 : \tau_2 \mid \phi_2 \\ \boxed{\begin{array}{c} \mathtt{t} = \mathit{typeof}(\mathcal{T}, e_1\ op_b\ e_2) \quad \mathit{isScalar}(\mathtt{t}) \implies \mathsf{scalar}(\tau_3) \\ \tau_1 \trianglelefteq \tau_3 \quad \tau_2 \trianglelefteq \tau_3 \quad \tau_3 \hookrightarrow_{0,|\mathtt{t}|} \tau_3 \quad \mathsf{collapsed}(\tau_3) \end{array}}^{\phi} \end{array}}{e_1\ op_b\ e_2 : \tau_3 \mid \phi_1 \wedge \phi_2 \wedge \phi}$$

Figure 3.6: Constraint generation rules

We only discuss some of the rules in detail. The rule Malloc generates the constraints for a `malloc` operation. We assume that each occurrence of `malloc` in the program is tagged with a unique identifier l and labeled with a unique cell variable τ representing the memory allocated by that `malloc`. The return value of `malloc` is a pointer with associated cell variable τ'. Thus, τ' points to τ.

The rules Dir-Sel, Arith-Op, and Cast are critical for the field-sensitive analysis. In particular, DIR-SEL generates constraints for field selections. A field `f` within a record or union expression e is associated with a cell variable τ_f. The rule states that there must be a contains-edge from the cell variable τ associated with e to τ_f with the appropriate offsets. Rule ARITH-OP is for arithmetic operations, which may involve pointer arithmetic. The cell variables τ_1, τ_2 and τ are associated with e_1, e_2, and $e_1\ op_b\ e_2$, respectively. Any cells pointed to by τ_1 and τ_2 must be equal, which is expressed by the constraints $\tau_1 \trianglelefteq \tau$ and $\tau_2 \trianglelefteq \tau$. Moreover, if τ points to another cell τ', then pointer arithmetic collapses all relevant cells containing τ', since we can no longer guarantee structured access to the memory represented by τ'. Rule CAST handles pointer cast operations. A pointer cast can change the points-to range of a pointer. In the rule, τ_1 and τ_2 represent the operand pointer and the result pointer, respectively. τ_1' and τ_2' represent their points-to contents. Similar to `malloc`, each pointer cast (`t`$*$) has a unique identifier l and is labeled with a unique cell variable τ_2' that represents the points-to content of the result pointer. The constraint $\mathsf{cast}(s, \tau_1', \tau_2')$ specifies that both τ_1' and τ_2' are within the same source containers with the same offsets. In particular, the size of τ_2' must be consistent with the size s of the type `t`.

3.2.3 Constraint Resolution

We next explain the constraint resolution step that computes a CFPG G from the generated constraint ϕ such that $G \models \phi$.

The resolution procedure must be able to reason about containment between cells, which is a transitive relation. Inspired by a decision procedure for the reachability relation in function graphs [27], we propose a rule-based procedure for this purpose.

The procedure is defined by a set of inference rules that infer new constraints from given constraints. The rules are shown in Figure 3.7. They are derived directly from the semantics of the constraints and the consistency properties of CFPGs. Some of the rules make use of the following syntactic short-hand

$$\mathsf{overlap}(\tau, \tau_1, i_1, j_1, \tau_2, i_2, j_2) \equiv \tau \hookrightarrow_{i_1,j_1} \tau_1 \land \tau \hookrightarrow_{i_2,j_2} \tau_2 \land i_1 \leq i_2 \land i_2 < j_1 \quad (3.15)$$

We omit the rules for reasoning about equality and inequality constraints, as they are straightforward. We also omit the rules for detecting conflicts. The only possible conflicts are inconsistent equality constraints such as $i = \top$ and inconsistent inequality constraints such as $i < \top$.

Our procedure maintains a *context* of constraints currently asserted to be true. The initial context is the set of constraints collected in the first phase. At each step, the rewrite rules are applied on the current context. For each rule, if the antecedent formulas are matched with formulas in the context, the consequent formula is added back to the context. The rules are applied until a conflict-free saturated context is obtained. The rule SPLIT branches on disjunctions. Note that the rules do not generate new disjunctions. All disjunctions come from the constraints of the form $i \preceq \eta$ and $i \sqsubseteq \eta$ in the initial context. Each disjunction in the initial context has at least one satisfiable branch. Our procedure uses a greedy heuristic that first chooses for each disjunction the branch that preserves more information and then backtracks on a conflict to choose the other branch. For example, for a disjunct $i \sqsubseteq \eta$, we first try $i = \eta$ before we choose $\eta = \top$.

Once a conflict-free saturated context has been derived, we construct the CFPG by using the equivalence classes of cell variables induced by the derived equality constraints as the cells of the graph. The other components of the graph can be constructed directly from the derived constraints.

Termination. To see that the procedure terminates, note that none of the rules introduce new cell variables τ. Moreover, the only rules that can increase the offsets i, j in containment constraints $\tau_1 \hookrightarrow_{i,j} \tau_2$ are CAST and TRANS. The application of these rules can be restricted in such a way that the offsets in the generated constraints do not exceed the maximal byte size of any of the types in the input program. With this restriction, the rules will only generated a bounded number of containment constraints.

$$\textsc{Size1}\ \frac{\tau_1 \hookrightarrow_{i,j} \tau_2 \quad \mathsf{size}(\tau_1) = k}{j \le k} \qquad \textsc{Size2}\ \frac{\tau_1 \hookrightarrow_{i,j} \tau_2 \quad \mathsf{size}(\tau_2) = k}{j - i = k}$$

$$\textsc{Refl}\ \frac{\mathsf{size}(\tau) = i}{\tau \hookrightarrow_{0,i} \tau} \qquad \textsc{Trans}\ \frac{\tau_1 \hookrightarrow_{i_1,j_1} \tau_2 \qquad \tau_2 \hookrightarrow_{i_2,j_2} \tau_3}{\tau_1 \hookrightarrow_{i_1+i_2,i_1+j_2} \tau_3}$$

$$\textsc{Source}\ \frac{\tau_1 \hookrightarrow_{i,j} \tau_2 \qquad \mathsf{source}(\tau_2)}{\tau_1 = \tau_2} \qquad \textsc{AntiSym}\ \frac{\tau_1 \hookrightarrow_{i_1,j_1} \tau_2 \qquad \tau_2 \hookrightarrow_{i_2,j_2} \tau_1}{\tau_1 = \tau_2}$$

$$\textsc{Collapse1}\ \frac{\tau_1 \hookrightarrow_{i,j} \tau_2 \qquad \mathsf{scalar}(\tau_1)}{\tau_1 = \tau_2} \qquad \textsc{Linear}\ \frac{\begin{array}{c}\tau \hookrightarrow_{i_1,j_1} \tau_1 \qquad \tau \hookrightarrow_{i_2,j_2} \tau_2 \\ i_1 \le i_2 < j_2 \le j_1\end{array}}{\tau_1 \hookrightarrow_{i_2-i_1,j_2-i_1} \tau_2}$$

$$\textsc{Overlap}\ \frac{\mathsf{scalar}(\tau_1) \quad \mathsf{scalar}(\tau_2) \quad \mathsf{overlap}(\tau, \tau_1, i_1, j_1, \tau_2, i_2, j_2)}{\tau_1 = \tau_2}$$

$$\textsc{Collapse2}\ \frac{\begin{array}{c}\mathsf{collapsed}(\tau) \quad \tau \rightharpoonup \tau_1 \quad \tau' \hookrightarrow_{i,j} \tau_2 \\ \mathsf{overlap}(\tau'', \tau_1, i_1, j_1, \tau_2, i_2, j_2)\end{array}}{\tau' = \tau_1} \qquad \textsc{Cast}\ \frac{\begin{array}{c}\mathsf{cast}(k, \tau_1, \tau_2) \qquad \mathsf{source}(\tau) \\ \tau \hookrightarrow_{i,j} \tau_1\end{array}}{\tau \hookrightarrow_{i,i+k} \tau_2}$$

$$\textsc{Collapse3}\ \frac{\begin{array}{c}\mathsf{collapsed}(\tau) \quad \tau \rightharpoonup \tau_1 \quad \tau' \hookrightarrow_{i,j} \tau_2 \\ \mathsf{overlap}(\tau'', \tau_2, i_2, j_2, \tau_1, i_1, j_1)\end{array}}{\tau' = \tau_1} \qquad \textsc{Scalar}\ \frac{\mathsf{size}(\tau) = \top}{\mathsf{scalar}(\tau)}$$

$$\textsc{Points}\ \frac{\tau \rightharpoonup \tau_1 \qquad \tau \rightharpoonup \tau_2}{\tau_1 = \tau_2} \qquad \textsc{PtrEq}\ \frac{\tau_1 \trianglelefteq \tau_2 \qquad \tau_1 \rightharpoonup \tau}{\tau_2 \rightharpoonup \tau} \qquad \textsc{Split}\ \frac{\phi_1 \vee \phi_2}{\phi_1 \qquad \phi_2}$$

Figure 3.7: Constraint resolution rules

3.3 Soundness

The soundness proof of the analysis is split into three steps. First, we prove that the CFPG resulting from the constraint resolution indeed satisfies the original constraints that are generated from the program. The proof shows that the inference rules are all consequences of the semantics of the constraints and the consistency properties of CFPGs. The second step defines an abstract semantics of programs in terms of abstract stores. These abstract stores only keep track of the partition of the byte-level memory into alias groups according to the computed CFPG. We then prove that the computed CFPG is a safe inductive invariant of the abstract semantics. The safety of the abstract semantics is defined in such a way that it guarantees that the computed CFPG describes a valid partition of the reachable program states into alias groups. Finally, we prove that the abstract semantics simulates the concrete byte-level semantics of programs.

3.3.1 Proof of Inference Rules

The proof of the inference rules in Fig. 3.7 are presented in this section. First, the rule [Split] is standard following from the logical meaning of $\vee$.

Rules [Refl], [Trans] and [AntiSym] state that $contains'$ is a reflexive, transitive and anti-symmetric relation. Their conclusions are obviously drawn from (3.1), (3.2) and (3.3). Rule [Linear] conforms to the linearity property of *contains* in (3.4).

Rule [Scalar] states that cells with unknown size must be scalar, following from (3.5). Rules [Source] and [Collapse1] conform to the property of source cells and scalar cells in the *contains* relation. The conclusions are drawn from

(3.7) and (3.8). Rule [Overlap] conforms to the property that two scalar cells are equivalent if they are overlapped stated in (3.11).

Rules [Size1] and [Size2] conform to the consistency of the contains relation. From $G \models \tau_1 \hookrightarrow_{i,j} \tau_2$, we have $\tau_1^G \overset{G}{\hookrightarrow}_{i,j} \tau_2^G$. From (3.9), we have $j \preceq size(\tau_1^G) \wedge j - i \sqsubseteq size(\tau_2^G)$. In [Size1], $G \models \mathsf{size}(\tau_1) = k$, we have $\mathsf{size}(\tau_1)^G = size(\tau_1^G) = k \in \mathbb{N}$. Then, $size(\tau_1^G) \neq \top$ and $j \leq k$. In [Size2], $G \models \mathsf{size}(\tau_2) = k$, we have $\mathsf{size}(\tau_2)^G = size(\tau_2^G) = k \in \mathbb{N}$. Then, $size(\tau_2^G) \neq \top$ and $j - i = k$.

Rule [Cast] conforms to the semantics of predicate $\mathsf{cast}(k, \tau_1, \tau_2)$ defined in (3.12). Rules [Collapse2] and [Collapse3] conform to the semantics of predicate $\mathsf{collapsed}(\tau)$ defined in (3.13). Consider the predicate $overlap^G$ is symmetric binary relation defined in (3.15), [Collapse2] draws the conclusion in one direction and [Collapse3] draws in the other direction.

Rule [Points] states the fact of points-to relation that two cells are equivalent if they are pointed by the same cell. Rule [PtrEq] states the equivalence of the points-to content according to the semantics of the predicate $\tau_1 \trianglelefteq \tau_2$ defined in (3.14).

3.3.2 Abstract Semantics

In this section, we introduce a abstract semantics of program in the language in Fig. 3.5, and prove that the CFPG computed in the constraints resolution is a safe inductive invariant.

3.3.2.1 Abstract States.

Recall that each program expression is assigned with a unique cell variable τ and program variables `x` are always assigned the same cell variable τ_x, where $\tau, \tau_x \in \mathbb{C}$.

The abstract semantics is defined in terms of the cell variables. Let Γ denote the environment mapping expressions to cell variables and G denote the computed CFPG graph. An *abstract state* $G^\sharp$ is a points-to graph built over a finite set of cell variables $\mathcal{C}$ bound with program expressions where $\mathcal{C} \subseteq \mathbb{C}$, denoted as a tuple $G^\sharp = (contains^\sharp, ptsto^\sharp, eqs^\sharp)$.

- $contains^\sharp \subseteq \mathcal{C} \times \mathbb{N} \times \mathbb{N} \times \mathcal{C}$ is a *containment relation* on cell variables,
- $ptsto^\sharp : \mathcal{C} \to \mathcal{C}$ is a abstract store tracking the points-to relation,
- $eqs^\sharp : \mathcal{C} \times \mathcal{C}$ is a *equivalent relation* on cell variables.

We use the notation $ptsto^\sharp[\tau \mapsto \tau']$ to represent the updated points-to relation is identical to $ptsto^\sharp$ except that the points-to cell variable of τ is τ'. The notation $G^\sharp[x := y]$ is used to represent the updated state is identical to $G^\sharp$ except the component x is updated to y.

The *initial* state is $G^\sharp_0 = (contains^\sharp_0, ptsto^\sharp_0, eqs^\sharp_0)$. The state component is subscripted by the subscript of the state to which it belongs, so $ptsto^\sharp_0$ represents the points-to relation of state $G^\sharp_0$. $ptsto^\sharp_0$ is an empty store, and both $contains^\sharp_0$ and $eqs^\sharp_0$ are empty sets.

Given an abstract state $G^\sharp = (contains^\sharp, ptsto^\sharp, eqs^\sharp)$, $G^{\sharp^+}$ is the saturated state derived from $G^\sharp$, according to the inference rules in Fig. 3.7

$$G^{\sharp^+} \stackrel{\mathsf{def}}{=} (contains^{\sharp^+}, ptsto^\sharp, eqs^{\sharp^+})$$

where $contains^{\sharp^+}$ and $eqs^{\sharp^+}$ are saturated sets of containment and equivalence relations, $contains^\sharp \subseteq contains^{\sharp^+}$ and $eqs^\sharp \subseteq eqs^{\sharp^+}$.

Lemma 1. *Given a program state $G^\sharp$ and a CFPG G, if $G^\sharp \sqsubseteq^\sharp G$, then $G^{\sharp^+} \sqsubseteq^\sharp G$.*

Proof. This lemma holds with respect to the soundness of those inference rules. $\square$

3.3.2.2 Expression Semantics.

The abstract semantics of an expression e is denoted as the state transition $\langle G_1^\sharp, e\rangle \Downarrow^\sharp G_2^\sharp$. There are two shortcut functions are referred in the semantics:

- $cast^\sharp(contains^\sharp, \tau_1, \tau_2, \mathtt{t})$ is used for the cast expression, deriving a set of containment relations:

$$\begin{aligned} cast^\sharp(contains^\sharp, \tau_1, \tau_2, \mathtt{t}) \;&\equiv \\ \{(\tau, i, i+|\mathtt{t}|, \tau_1) \mid (\tau, i, j, \tau_2) &\in contains^\sharp, \mathsf{source}(\tau)\} \end{aligned} \tag{3.16}$$

  where $\tau_1, \tau_2 \in \mathcal{C}$ and $\mathtt{t}$ is the type to cast. For any τ, if $\mathsf{source}(\tau)$ and $\tau \hookrightarrow_{i,j} \tau_2 \in contains^\sharp$, we can derive $\tau \hookrightarrow_{i,i+|\mathtt{t}|} \tau_1$.

- $collapse^\sharp(contains^\sharp, \tau)$ is used for the pointer arithmetic operations, deriving a set of equivalence relations:

$$\begin{aligned} & collapse^\sharp(contains^\sharp, \tau) \;\equiv \\ & \left\{ (\tau, \tau_c') \;\middle|\; \begin{aligned} & (\tau_c, i_1, i_2, \tau), (\tau_c, j_1, j_2, \tau'), (\tau_c', k_1, k_2, \tau') \in contains^\sharp, \\ & i_1 \le j_1 < i_2 \vee j_1 \le i_1 < j_2 \end{aligned} \right\} \end{aligned} \tag{3.17}$$

  where $\tau \in \mathcal{C}$. If τ and τ' are overlapping within cell variable τ_c, we can derive that τ is equivalent with any cell variable τ_c' that contains τ'.

▶ **Constant**

$$\textsc{Const}\ \frac{}{\langle G^\sharp, c\rangle \Downarrow^\sharp G^\sharp}$$

▶ **Variable**

$$\textsc{Var}\ \frac{}{\langle G^\sharp, \mathtt{x}\rangle \Downarrow^\sharp G^\sharp}$$

▶ **Sequence**

$$\textsc{Seq}\ \frac{\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp \quad \langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp}{\langle G^\sharp, e_1, e_2\rangle \Downarrow^\sharp G_2^\sharp}$$

▶ **Address of**

$$\textsc{Addr}\ \frac{\langle G^\sharp, e\rangle \Downarrow^\sharp G_1^\sharp}{\langle G^\sharp, \&e\rangle \Downarrow^\sharp G_2^\sharp}$$

$$\text{where} \quad \tau = \Gamma(\&e), \tau' = \Gamma(e), G_2^\sharp = G_1^\sharp[\mathit{ptsto}^\sharp \coloneqq \mathit{ptsto}_1^\sharp[\tau \mapsto \tau']]^+.$$

▶ **Dereference**

$$\textsc{Deref}\ \frac{\langle G^\sharp, e\rangle \Downarrow^\sharp G_1^\sharp}{\langle G^\sharp, *e\rangle \Downarrow^\sharp G_2^\sharp}$$

$$\text{where} \quad \tau = \Gamma(e), \tau' = \Gamma(*e), G_2^\sharp = G_1^\sharp[\mathit{ptsto}^\sharp \coloneqq \mathit{ptsto}_1^\sharp[\tau \mapsto \tau']]^+.$$

▶ Field selection

$$\textsc{FieldSel}\ \frac{\langle G^\sharp, e\rangle \Downarrow^\sharp G_1^\sharp}{\langle G^\sharp, e.\texttt{f}\rangle \Downarrow^\sharp G_2^\sharp}$$

$$\begin{aligned}\text{where}\quad & \tau = \Gamma(e), \tau_f = \Gamma(e.\texttt{f}), o = \mathit{offset}(\texttt{t}, \texttt{f}),\\ & G_2^\sharp = G_1^\sharp[\mathit{contains}^\sharp := \mathit{contains}_1^\sharp \cup \{(\tau, o, o + |\texttt{t.f}|, \tau_f)\}]^+.\end{aligned}$$

▶ Malloc

$$\textsc{Malloc}\ \frac{\langle G^\sharp, e\rangle \Downarrow^\sharp G_1^\sharp}{\langle G^\sharp, (\texttt{t}*)\mathsf{malloc}_l(e)\rangle \Downarrow^\sharp G_2^\sharp}$$

$$\text{where}\quad \tau_1 = \Gamma((\texttt{t}*)\mathsf{malloc}_l(e)), \tau_2 = \Gamma(\mathsf{malloc}_l), G_2^\sharp = G_1^\sharp[\mathit{ptsto}^\sharp := \mathit{ptsto}_1^\sharp[\tau_1 \mapsto \tau_2]]^+.$$

▶ Cast

$$\textsc{Cast}\ \frac{\langle G^\sharp, e\rangle \Downarrow^\sharp G_1^\sharp}{\langle G^\sharp, (\texttt{t}*)_l e\rangle \Downarrow^\sharp G_2^\sharp}$$

$$\begin{aligned}\text{where}\quad & \tau_1 = \Gamma((\texttt{t}*)_l\ e), \tau_2 = \Gamma(l), \tau_1' = \Gamma(e), \tau_2' = \mathit{ptsto}_1^\sharp(\tau_1'),\\ & G_2^\sharp = G_1^\sharp\begin{bmatrix} \mathit{ptsto}^\sharp := \mathit{ptsto}_1^\sharp[\tau_1 \mapsto \tau_2] \\ \mathit{contains}^\sharp := \mathit{contains}_1^\sharp \cup \mathit{cast}^\sharp(\mathit{contains}_1^\sharp, \tau_2, \tau_2', \texttt{t}) \end{bmatrix}^+.\end{aligned}$$

▶ Assignment

$$\textsc{Assg}_1 \frac{\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp \quad \langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp \quad \tau_2 \notin \mathtt{dom}(ptsto_2^\sharp)}{\langle G^\sharp, e_1 = e_2\rangle \Downarrow^\sharp G_3^\sharp}$$

where $\quad \tau = \Gamma(e_1 = e_2), \tau_2 = \Gamma(e_2), G_3^\sharp = G_2^\sharp[eqs^\sharp := eqs_2^\sharp \cup \{(\tau, \tau_2)\}]^+.$

$$\textsc{Assg}_2 \frac{\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp \quad \langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp \quad \tau_2 \in \mathtt{dom}(ptsto_2^\sharp)}{\langle G^\sharp, e_1 = e_2\rangle \Downarrow^\sharp G_3^\sharp}$$

where $\quad \tau = \Gamma(e_1 = e_2), \tau_1 = \Gamma(e_1), \tau_2 = \Gamma(e_2), \tau_2' = ptsto_2^\sharp(\tau_2),$

$$G_3^\sharp = G_2^\sharp \left[\begin{array}{l} ptsto^\sharp := ptsto_2^\sharp[\tau_1 \mapsto \tau_2', \tau \mapsto \tau_2'] \\ eqs^\sharp := eqs_2^\sharp \cup \{(\tau, \tau_2)\} \end{array} \right]^+.$$

▶ Binary Operation

$$\textsc{Arith-Op}_1 \frac{\begin{array}{c} \langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp \quad \langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp \\ \tau_1 = \Gamma(e_1) \quad \tau_2 = \Gamma(e_2) \quad \tau_1, \tau_2 \notin \mathtt{dom}(ptsto_2^\sharp) \end{array}}{\langle G^\sharp, e_1 \ op_b \ e_2\rangle \Downarrow^\sharp G_2^\sharp}$$

$$\textsc{Arith-Op}_2\ \frac{\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp \quad \langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp \quad \tau_1 \in \mathtt{dom}(ptsto_2^\sharp) \quad \tau_2 \notin \mathtt{dom}(ptsto_2^\sharp)}{\langle G^\sharp, e_1\ op_b\ e_2\rangle \Downarrow^\sharp G_3^\sharp}$$

$$\text{where}\quad \tau_1 = \Gamma(e_1), \tau_2 = \Gamma(e_2), \tau = \Gamma(e_1\ op_b\ e_2), \tau_1' = ptsto_2^\sharp(\tau_1),$$

$$G_3^\sharp = G_2^\sharp \begin{bmatrix} eqs^\sharp \coloneqq eqs_2^\sharp \cup collapse^\sharp(contains_2^\sharp, \tau_1') \\ ptsto^\sharp \coloneqq ptsto_2^\sharp[\tau \mapsto \tau_1'] \end{bmatrix}^+$$

$$\textsc{Arith-Op}_3\ \frac{\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp \quad \langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp \quad \tau_1 \notin \mathtt{dom}(ptsto_2^\sharp) \quad \tau_2 \in \mathtt{dom}(ptsto_2^\sharp)}{\langle G^\sharp, e_1\ op_b\ e_2\rangle \Downarrow^\sharp G_3^\sharp}$$

$$\text{where}\quad \tau_1 = \Gamma(e_1), \tau_2 = \Gamma(e_2), \tau = \Gamma(e_1\ op_b\ e_2), \tau_2' = ptsto_2^\sharp(\tau_2),$$

$$G_3^\sharp = G_2^\sharp \begin{bmatrix} eqs^\sharp \coloneqq eqs_2^\sharp \cup collapse^\sharp(contains_2^\sharp, \tau_2') \\ ptsto^\sharp \coloneqq ptsto_2^\sharp[\tau \mapsto \tau_2'] \end{bmatrix}^+$$

$$\textsc{Arith-Op}_4\ \frac{\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp \quad \langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp \quad \tau_1 \in \mathtt{dom}(ptsto_2^\sharp) \quad \tau_2 \in \mathtt{dom}(ptsto_2^\sharp)}{\langle G^\sharp, e_1\ op_b\ e_2\rangle \Downarrow^\sharp G_3^\sharp}$$

$$\text{where}\quad \tau_1 = \Gamma(e_1), \tau_2 = \Gamma(e_2), \tau = \Gamma(e_1\ op_b\ e_2), \tau_1' = ptsto_2^\sharp(\tau_1), \tau_2' = ptsto_2^\sharp(\tau_2),$$

$$G_3^\sharp = G_2^\sharp \begin{bmatrix} ptsto^\sharp \coloneqq ptsto_2^\sharp[\tau \mapsto \tau_1'] \\ eqs^\sharp \coloneqq eqs_2^\sharp \cup \{(\tau_1', \tau_2')\} \cup \\ \qquad collapse^\sharp(contains_2^\sharp, \tau_1') \cup collapse^\sharp(contains_2^\sharp, \tau_2') \end{bmatrix}^+$$

3.3.2.3 Invariant

Given a state graph $G^\sharp$ and a computed CFPG G, we define a partial order $G^\sharp \sqsubseteq^\sharp G$. Recall that within G, there is a component *cell* mapping cell variables to cells, where given $\tau \in \mathbb{C}$, $cell(\tau) = \tau^G \in C$. This partial order requires *cell* is a homomorphism mapping that the *points-to*, *containment* and *equivalent* relations in $G^\sharp$ are preserved in G.

Definition 1. *Given a state* $G^\sharp = (contains^\sharp, ptsto^\sharp, eqs^\sharp)$ *and a CFPG graph* G*, we say* $G^\sharp \sqsubseteq^\sharp G$ *if (1)* $\forall \tau \in \mathtt{dom}(ptsto^\sharp)\ .\ \tau^G \stackrel{G}{\rightharpoonup} ptsto^\sharp(\tau)^G$*; (2)* $\forall (\tau_1, i, j, \tau_2) \in contains^\sharp \implies \tau_1^G \stackrel{G}{\hookrightarrow}_{i,j} \tau_2^G$*; (3)* $\forall (\tau_1, \tau_2) \in eqs^\sharp \implies \tau_1^G = \tau_2^G$.

Theorem 1. *Given a program* e*, if* $\langle G_0^\sharp, e\rangle \Downarrow^\sharp G_n^\sharp$*, then the computed CFPG graph* G *is an inductive invariant of the program such that* $G_0^\sharp \sqsubseteq^\sharp G$ *and* $G_n^\sharp \sqsubseteq^\sharp G$.

3.3.3 Concrete Semantics

In this section, we define the concrete semantics of the program, and setup a correspondence between the concrete and abstract semantics ensuring the aliasing and overlapping between memory areas in the concrete semantics are captured in the abstract semantics.

Let $\mathbb{X}$ denote a set of variables and $\mathbb{V}$ denote a set of values. $\mathbb{V}_{\mathtt{t}}$ is a set of values with scalar type `t`. Pointer values is defined as a set of memory locations: $\mathbb{V}_{\mathtt{ptr}} = \{(a, i) \mid a \in \mathbb{A}, i \in \mathbb{N}\} \cup \{\mathtt{NULL}\}$, where $\mathbb{A}$ is a set of base addresses. Each pointer is either a pair of a base address and a byte offset or a null pointer `NULL`. Let $\mathbb{B} = \mathbb{V}_{\mathtt{ptr}} \times \mathbb{N}$ denote a set of memory blocks, and each memory block is represented by a pair of a starting location and a size. A few predicates over memory blocks are introduced here:

- $\texttt{overlap}(b_1, b_2)$, indicating that b_1 and b_2 are overlapping

$$\exists \left(\begin{array}{c} a \in \mathbb{A}, \\ i_1, i_2, s_1, s_2 \in \mathbb{N} \end{array} \right) . \left(\begin{array}{l} b_1 = ((a, i_1), s_1) \wedge \\ b_2 = ((a, i_2), s_2) \end{array} \right) \wedge \left(\begin{array}{l} i_1 < i_2 < i_1 + s_1 \vee \\ i_2 < i_1 < i_2 + s_2 \end{array} \right)$$

- $\texttt{icontains}(b_1, i, j, b_2)$, indicating b_1 contains b_2 within interval $[i, j)$

$$\exists \left(\begin{array}{c} a \in \mathbb{A}, \\ i_1, i_2, s_1, s_2 \in \mathbb{N} \end{array} \right) . \left(\begin{array}{l} b_1 = ((a, i_1), s_1) \wedge \\ b_2 = ((a, i_2), s_2) \end{array} \right) \wedge \left(\begin{array}{l} i = i_2 - i_1 \wedge \\ j = i + s_2 \leq i_1 + s_1 \end{array} \right)$$

- $\texttt{contains}(b_1, b_2)$, indicating that b_1 contains b_2

$$\exists i, j \in \mathbb{N} \,.\, \texttt{icontains}(b_1, i, j, b_2)$$

3.3.3.1 Concrete State.

We let $\mathbb{M}$ denote the set of memory states. The memory read/write should take a memory block as a parameter that either read or write a value exactly into the memory block, which are defined as: $\mathsf{read} : \mathbb{M} \times \mathbb{B} \to \mathbb{V}$ and $\mathsf{write} : \mathbb{M} \times \mathbb{B} \times \mathbb{V} \to \mathbb{M}$.

The *concrete state* $G^\natural$ is denoted as a tuple

$$G^\natural = (\epsilon, ptsto^\natural, src, type, part, valid, m, \mathcal{B})$$

- $\epsilon : \mathbb{X} \to \mathbb{A}$, a mapping variables to *distinct* base addresses;
- $ptsto^\natural : \mathbb{B} \to \mathbb{B}$, tracking the points-to relation between memory blocks;
- $src : \mathbb{B} \to \mathbb{B}$, mapping a memory block to its source block;

- $type : \mathbb{B} \rightarrow$ type, mapping a memory block to its C type;
- $part : \mathbb{C} \rightarrow \mathcal{P}(\mathbb{B})$, mapping a cell variable to a set of memory blocks;
- $valid : \mathbb{B} \rightarrow \texttt{type}$, mapping a valid memory block to its *scalar* access type;
- $m \in \mathbb{M}$, a concrete store simulating the memory;
- $\mathcal{B} \subseteq \mathbb{B}$, a collection of memory blocks;

We use the notation $f[x \mapsto y]$ to represent the updated map f' is identical to f except that $f(x) = y$. The notation $part[\tau \Leftarrow b]$ is used to represent $part$ is updated by adding block b to the block set of τ. This notation is equivalent to $part[\tau \mapsto part(\tau) \cup \{b\}]$. The notation $G^\natural[x := y]$ is used to represent the updated state is identical to $G^\natural$ except the component x is updated to y.

Given a concrete state $G^\natural$, function $\texttt{isValid}(G^\natural, b)$ is introduced to tell if $b \in \mathbb{B}$ is contained within a valid block in $\texttt{dom}(valid)$:

$$\texttt{isValid}(G^\natural, b) \;\; \equiv \;\; \exists b' \in \texttt{dom}(valid) \; . \; \texttt{contains}(b', b)$$

The *initial* state is

$$G_0^\natural = (\epsilon, ptsto_0^\natural, src_0, type_0, part_0, valid_0, m_0, \mathcal{B}_0)$$

where each state component is subscripted by the subscript of the state to which it belongs, so m_0 represents the memory state of state $G_0^\natural$. In the initial state, for $\texttt{x} \in \mathbb{X}$, let $a_x = \epsilon(\texttt{x})$, $\texttt{t}_x = typeof(\mathcal{T}, \texttt{x})$, $b_x = (a_x, |\texttt{t}_x|)$ and $\tau_x = \Gamma(\texttt{x})$. Then $src_0(b_x) = b_x$, $type_0(b_x) = \texttt{t}_x$, $part_0(\tau_x) = \{b_x\}$, $valid_0(b_x) = \texttt{t}_x$, and $\mathcal{B}_0 = \{b_x \mid \texttt{x} \in \mathbb{X}\}$. m_0 is a fresh array variable and $ptsto_0^\natural$ is empty.

$$\textsc{Non-Scalar}\ \frac{\langle G_1^\natural, e\rangle \Downarrow_l^\natural \langle loc, G_2^\natural\rangle \quad \mathtt{t} = \mathit{typeof}(\mathcal{T}, e) \quad \mathtt{t} \text{ is not scalar}}{\langle G_1^\natural, e\rangle \Downarrow_r^\natural \langle loc, G_2^\natural\rangle}$$

$$\textsc{Scalar}\ \frac{\langle G_1^\natural, e\rangle \Downarrow_l^\natural \langle loc, G_2^\natural\rangle \quad \mathtt{t} = \mathit{typeof}(\mathcal{T}, e) \quad \mathtt{t} \text{ is scalar}}{\langle G_1^\natural, e\rangle \Downarrow_r^\natural \langle \mathsf{read}(m_2, loc, |\mathtt{t}|), G_2^\natural\rangle}$$

Figure 3.8: Right-value evaluation in concrete semantics

3.3.3.2 Concrete Semantics.

The evaluation of expression is defined by two judgements:

- $\langle G_1^\natural, e\rangle \Downarrow_l^\natural \langle loc, G_2^\natural\rangle$ left-value evaluation of expression, where loc is the left-value of expression e;
- $\langle G_1^\natural, e\rangle \Downarrow_r^\natural \langle v, G_2^\natural\rangle$ right-value evaluation of expression, where v is the right-value of expression e.

In many contexts, the semantics requires left-values to become right-values of a given expression. The right-value evaluation can be inferred from the left-value as in Fig. 3.8.

▶ Constant

$$\textsc{Const}\ \frac{}{\langle G^\natural, c\rangle \Downarrow_r^\natural \langle c, G^\natural\rangle}$$

▶ Variable

$$\textsc{Var}\ \frac{}{\langle G^\natural, \mathtt{x}\rangle \Downarrow_l^\natural \langle (\epsilon(\mathtt{x}), 0), G^\natural\rangle}$$

▶ Dereference

$$\textsc{Deref}\ \frac{\langle G^\natural, e\rangle \Downarrow_l^\natural \langle loc, G_1^\natural\rangle \quad \langle G^\natural, e\rangle \Downarrow_r^\natural \langle v, G_1^\natural\rangle \quad v \in \mathbb{V}_{\texttt{ptr}} \\ \texttt{t} = \mathit{typeof}(\mathcal{T}, *e) \quad b = (loc, |\texttt{ptr}|) \quad b_* = (v, |\texttt{t}|) \quad \mathit{ptsto}_1^\natural(b) = b_*}{\langle G^\natural, *e\rangle \Downarrow_l^\natural \langle v, G_1^\natural\rangle}$$

▶ Address of

$$\textsc{Addr}\ \frac{\langle G^\natural, e\rangle \Downarrow_l^\natural \langle loc, G_1^\natural\rangle \quad loc_\& = (a, 0) \quad a \text{ is fresh}}{\langle G^\natural, \&e\rangle \Downarrow_l^\natural \langle loc_\&, G_2^\natural\rangle}$$

where $\texttt{t} = \mathit{typeof}(\mathcal{T}, e), b = (loc, |\texttt{t}|), b_\& = (loc_\&, |\texttt{ptr}|), \tau = \Gamma(\&e),$

$$G_2^\natural = G_1^\natural \begin{bmatrix} \mathit{ptsto}^\natural := \mathit{ptsto}_1^\natural[b_\& \mapsto b], & \mathit{src} := \mathit{src}_1[b_\& \mapsto b_\&], \\ \mathit{type} := \mathit{type}_1[b_\& \mapsto \texttt{ptr}], & \mathit{part} := \mathit{part}_1[\tau \Leftarrow b_\&], \\ m := \mathsf{write}(m_1, b_\&, loc), & \mathcal{B} := \mathcal{B}_1 \cup \{b_\&\} \end{bmatrix}.$$

▶ Sequence

$$\textsc{Seq}\ \frac{\langle G^\natural, e_1\rangle \Downarrow_l^\natural \langle loc_1, G_1^\natural\rangle \quad \langle G_1^\natural, e_2\rangle \Downarrow_l^\natural \langle loc_2, G_2^\natural\rangle}{\langle G^\natural, e_1, e_2\rangle \Downarrow_l^\natural \langle loc_2, G_2^\natural\rangle}$$

▶ Field selection

$$\textsc{FieldSel}\ \frac{\langle G^\natural, e\rangle \Downarrow_l^\natural \langle loc, G_1^\natural\rangle}{\langle G^\natural, e\rangle \Downarrow_l^\natural \langle loc_f, G_2^\natural\rangle}$$

where $\quad \mathtt{t} = \mathit{typeof}(\mathcal{T}, e), loc = (a, i), loc_f = (a, i + \mathit{offset}(\mathtt{t}, \mathtt{f})), \tau_f = \Gamma(e.\mathtt{f}),$

$b = (loc, |\mathtt{t}|), b_f = (loc_f, |\mathtt{t.f}|),$

$$G_2^\natural = G_1^\natural \begin{bmatrix} src \coloneqq src_1[b_f \mapsto src_1(b)], & type \coloneqq type_1[b_f \mapsto \mathtt{t.f}], \\ part \coloneqq part_1[\tau_f \Leftarrow b_f], & \mathcal{B} \coloneqq \mathcal{B}_1 \cup \{b_f\} \end{bmatrix}.$$

▶ Pointer Arithmetic Operation

$$\textsc{Arith-Op}_2\ \frac{\begin{array}{c}\langle G^\natural, e_1\rangle \Downarrow_l^\natural \langle loc_1, G_1^\natural\rangle \quad \langle G^\natural, e_1\rangle \Downarrow_r^\natural \langle v_1, G_1^\natural\rangle \quad \langle G_1^\natural, e_2\rangle \Downarrow_r^\natural \langle v_2, G_2^\natural\rangle \\ v_1 = (a, i) \in \mathbb{V}_{\mathtt{ptr}} \quad v_2 \notin \mathbb{V}_{\mathtt{ptr}} \quad b_1 = (loc_1, |\mathtt{ptr}|) \in \mathtt{dom}(ptsto_2^\natural) \\ \mathtt{t} = \mathit{typeof}(\mathcal{T}, *(e_1\ \pm\ e_2)) \quad v_2\%|\mathtt{t}| = 0 \\ loc = (a, i \pm v_2) \quad b'_* = (loc, |\mathtt{t}|) \quad \mathtt{isValid}(G_2^\natural, b_*) \quad \mathtt{isValid}(G_2^\natural, b'_*) \\ loc' = (a', 0) \quad a' \text{ is fresh}\end{array}}{\langle G^\natural, e_1 \pm e_2\rangle \Downarrow_l^\natural \langle loc', G_3^\natural\rangle}$$

where $\quad b' = (loc', |\mathtt{ptr}|), b_* = (v_1, |\mathtt{t}|) = ptsto_2^\natural(b_1), \tau = \Gamma(e_1\ \pm\ e_2),$

$$G_3^\natural = G_2^\natural \begin{bmatrix} src \coloneqq \mathtt{ite}(b'_* \in \mathtt{dom}(src_2), src_2[b' \mapsto b'], src_2[b' \mapsto b', b'_* \mapsto b'_*]) & \\ part \coloneqq part_2[\tau \Leftarrow b', \{\tau_* \Leftarrow b'_* \mid b_* \in part_2(\tau_*)\}] & \\ type \coloneqq type_2[b' \mapsto \mathtt{ptr}, b'_* \mapsto \mathtt{t}] & m \coloneqq \mathsf{write}(m_2, b', loc') \\ ptsto^\natural = ptsto_2^\natural[b' \mapsto b'_*] & \mathcal{B} \coloneqq \mathcal{B}_2 \cup \{b', b'_*\} \end{bmatrix}.$$

▶ Binary Operation

$$\textsc{Arith-Op}_1\ \frac{\langle G^\natural, e_1\rangle \Downarrow_r^\natural \langle v_1, G_1^\natural\rangle \quad \langle G_1^\natural, e_2\rangle \Downarrow_r^\natural \langle v_2, G_2^\natural\rangle \qquad v_1, v_2 \notin \mathbb{V}_{\texttt{ptr}} \quad loc = (a, 0) \quad a \text{ is fresh}}{\langle G^\natural, e_1\ op_b\ e_2\rangle \Downarrow_l^\natural \langle loc, G_3^\natural\rangle}$$

where $\quad \texttt{t} = \textit{typeof}(\mathcal{T}, e_1\ op_b\ e_2), b = (loc, |\texttt{t}|), \tau = \Gamma(e_1\ op_b\ e_2),$

$$G_3^\natural = G_2^\natural \begin{bmatrix} part := part_2[\tau \Leftarrow b] & type := type_2[b \mapsto \texttt{t}] \\ m := \mathsf{write}(m_2, b, v_1\ op_b\ v_2) & \mathcal{B} := \mathcal{B}_2 \cup \{b\} \end{bmatrix}.$$

▶ Malloc

$$\textsc{Malloc}\ \frac{\langle G^\natural, e\rangle \Downarrow_r^\natural \langle v, G_1^\natural\rangle \quad |\texttt{t}| \le v \qquad loc_l = (a_l, 0) \quad loc = (a, 0) \quad a, a_l \text{ are fresh}}{\langle G^\natural, (\texttt{t}*)\mathsf{malloc}_l(e)\rangle \Downarrow_l^\natural \langle loc_l, G_2^\natural\rangle}$$

where $\quad b_l = (loc_l, |\texttt{ptr}|), b_0 = (loc, |\texttt{t}|), b = (loc, v), \tau_1 = \Gamma((\texttt{t}*)\mathsf{malloc}_l(e)), \tau_2 = \Gamma(\mathsf{malloc}_l),$

$$G_2^\natural = G_1^\natural \begin{bmatrix} type = type_1[b_l \mapsto \texttt{ptr}, b_0 \mapsto \texttt{t}] \quad src = src_1[b_l \mapsto b_l, b_0 \mapsto b_0] \\ part = part_1[\tau_1 \Leftarrow b_l, \tau_2 \Leftarrow b_0] \quad ptsto^\natural = ptsto_1^\natural[b_l \mapsto b_0] \\ m = \mathsf{write}(m_1, b_l, loc)\ \ valid = valid_1[b_0 \mapsto \texttt{t}]\ \ \mathcal{B} = \mathcal{B}_1 \cup \{b_l, b_0\} \end{bmatrix}.$$

▶ **Cast**

$$\textsc{Cast}_1\ \frac{\begin{array}{c}\langle G^\natural, e\rangle \Downarrow_l^\natural \langle loc, G_1^\natural\rangle \quad \langle G^\natural, e\rangle \Downarrow_r^\natural \langle v, G_1^\natural\rangle \quad v \in \mathbb{V}_{\mathtt{ptr}} \\ a_l \text{ is fresh} \quad loc_l = (a_l, 0) \\ b = (loc, |\mathtt{ptr}|) \quad b_* = ptsto_1^\natural(b) = (v, s) \quad b'_* = (v, |\mathtt{t}|) \\ \mathtt{isValid}(G_1^\natural, b_*) \quad \mathtt{isValid}(G_1^\natural, b'_*) \quad \mathtt{contains}(src_1(b_*), b'_*)\end{array}}{\langle G^\natural, (\mathtt{t}*)_l\ e\rangle \Downarrow_l^\natural \langle loc_l, G_2^\natural\rangle}$$

where $\quad b_l = (loc_l, |\mathtt{ptr}|), \tau_1 = \Gamma((\mathtt{t}*)_l\ e), \tau_2 = \Gamma(l),$

$$G_2^\natural = G_1^\natural \begin{bmatrix} src \coloneqq src_1[b_l \mapsto b_l, b'_* \mapsto src_1(b_*)] & ptsto^\natural \coloneqq ptsto_1^\natural[b_l \mapsto b'_*] \\ type \coloneqq type_1[b_l \mapsto \mathtt{ptr}, b'_* \mapsto \mathtt{t}] & part \coloneqq part_1[\tau_1 \Leftarrow b_l, \tau_2 \Leftarrow b'_*] \\ m \coloneqq \mathsf{write}(m_1, b_l, \mathsf{convert}(v, \mathtt{t})) & \mathcal{B} \coloneqq \mathcal{B}_1 \cup \{b_l, b'_*\} \end{bmatrix}.$$

$$\textsc{Cast}_2\ \frac{\begin{array}{c}\langle G^\natural, e\rangle \Downarrow_l^\natural \langle loc, G_1^\natural\rangle \quad \langle G^\natural, e\rangle \Downarrow_r^\natural \langle v, G_1^\natural\rangle \quad v \in \mathbb{V}_{\mathtt{ptr}} \\ a_l \text{ is fresh} \quad loc_l = (a_l, 0) \\ b = (loc, |\mathtt{ptr}|) \quad b_* = ptsto_1^\natural(b) = (v, s) \quad b'_* = (v, |\mathtt{t}|) \\ \mathtt{isValid}(G_1^\natural, b_*) \quad \mathtt{isValid}(G_1^\natural, b'_*) \quad \neg\mathtt{contains}(src_1(b_*), b'_*)\end{array}}{\langle G^\natural, (\mathtt{t}*)_l\ e\rangle \Downarrow_l^\natural \langle loc_l, G_2^\natural\rangle}$$

where $\quad b_l = (loc_l, |\mathtt{ptr}|), \tau_1 = \Gamma((\mathtt{t}*)_l\ e), \tau_2 = \Gamma(l),$

$$G_2^\natural = G_1^\natural \begin{bmatrix} src \coloneqq \mathtt{ite}(b'_* \in \mathtt{dom}(src_1), src_1[b_l \mapsto b_l], src_1[b_l \mapsto b_l, b'_* \mapsto b'_*]) \\ part \coloneqq part_1[\tau_1 \Leftarrow b_l, \tau_2 \Leftarrow b'_*] \quad ptsto^\natural \coloneqq ptsto_1^\natural[b_l \mapsto b'_*] \\ type \coloneqq type_1[b_l \mapsto \mathtt{ptr}, b'_* \mapsto \mathtt{t}] \quad \mathcal{B} \coloneqq \mathcal{B}_1 \cup \{b_l, b'_*\} \\ m \coloneqq \mathsf{write}(m_1, b_l, \mathsf{convert}(v, \mathtt{t})) \end{bmatrix}.$$

▶ **Assignment**

$$\textsc{Assg}_1 \frac{\langle G^\natural, e_1\rangle \Downarrow_l^\natural \langle loc_1, G_1^\natural\rangle \quad \langle G_1^\natural, e_2\rangle \Downarrow_l^\natural \langle loc_2, G_2^\natural\rangle \quad \langle G_1^\natural, e_2\rangle \Downarrow_r^\natural \langle v, G_2^\natural\rangle \qquad b_1 = (loc_1, |\mathtt{t}|) \quad b_2 = (loc_2, |\mathtt{t}|) \quad b_2 \notin \mathtt{dom}(ptsto_2^\natural)}{\langle G^\natural, e_1 =_{\mathtt{t}} e_2\rangle \Downarrow_l^\natural \langle loc_1, G_2^\natural[m := \mathsf{write}(m_2, b_1, v)]\rangle}$$

$$\textsc{Assg}_2 \frac{\langle G^\natural, e_1\rangle \Downarrow_l^\natural \langle loc_1, G_1^\natural\rangle \quad \langle G_1^\natural, e_2\rangle \Downarrow_l^\natural \langle loc_2, G_2^\natural\rangle \quad \langle G_1^\natural, e_2\rangle \Downarrow_r^\natural \langle v, G_2^\natural\rangle \qquad b_1 = (loc_1, |\mathtt{t}|) \quad b_2 = (loc_2, |\mathtt{t}|) \quad b_2 \in \mathtt{dom}(ptsto_2^\natural)}{\langle G^\natural, e_1 =_{\mathtt{t}} e_2\rangle \Downarrow_l^\natural \langle loc_1, G_3^\natural\rangle}$$

where $G_3^\natural = G_2^\natural[ptsto^\natural := ptsto_2^\natural[b_1 \mapsto ptsto_2^\natural(b_2)], m := \mathsf{write}(m_2, b_1, v)]$.

3.3.3.3 Correspondence.

The abstract semantics is built over cell variables, while the concrete semantics is built over memory blocks. Function *part* connects them by mapping each cell variable to a collection of memory blocks.

At each step of program execution, there is an abstract state $G^\sharp$ and a concrete state $G^\natural$, and their correspondence is stated in lemma 2, 3, 4 and 5. We say $G^\natural$ conforms to $G^\sharp$, denoted as $G^\natural \sqsubseteq^\natural G^\sharp$, if these lemmas hold.

Lemma 2. *Given a concrete state $G^\natural = (\epsilon, ptsto^\natural, src, type, part, valid, m, \mathcal{B})$ and a abstract state $G^\sharp = (contains^\sharp, ptsto^\sharp, eqs^\sharp)$ at a program point, we have (1) the size of each memory block must be consistent with the cell size of any of its cell variables; (2) the cell variables associated with the same memory block must be* *equivalent.*

$$\forall b \in \mathcal{B}\ \forall \tau_1, \tau_2 \in \mathbb{C}\ .\ b \in part(\tau_1) \wedge b \in part(\tau_2) \implies (\tau_1, \tau_2) \in eqs^\sharp \tag{Eq1}$$

$$\forall b \in \mathcal{B}\ \forall \tau \in \mathbb{C}\ \exists a \in \mathbb{A}\ \exists s \in \mathbb{N}\ .\ b \in part(\tau) \wedge b = (a, s) \implies s \sqsubseteq \mathsf{size}(\tau) \tag{Eq2}$$

Lemma 3. *Given a concrete state $G^\natural = (\epsilon, ptsto^\natural, src, type, part, valid, m, \mathcal{B})$ and a abstract state $G^\sharp = (contains^\sharp, ptsto^\sharp, eqs^\sharp)$ at a program point, for any two memory blocks b_1 and b_2, if the concrete points-to relation holds between them, then the abstract points-to relation holds between their cell variables:*

$$\begin{aligned}&\forall b_1, b_2 \in \mathcal{B}\ \exists \tau_1, \tau_2 \in \mathbb{C}\ .\\ &\qquad b_1 \in part(\tau_1) \wedge b_2 \in part(\tau_2) \wedge b_1 = ptsto^\natural(b_2) \implies \tau_1 = ptsto^\sharp(\tau_2)\end{aligned}$$

Lemma 4. *Given a concrete state $G^\natural = (\epsilon, ptsto^\natural, src, type, part, valid, m, \mathcal{B})$ and a abstract state $G^\sharp = (contains^\sharp, ptsto^\sharp, eqs^\sharp)$ at a program point, for any two memory blocks $b_1, b_2 \in \mathcal{B}$, if b_1 contains b_2, we have either the abstract contains relation holds between their cell variables or their cell variables are equivalent (collapsed records or unions)*

$$\begin{aligned}&\forall \tau_1, \tau_2 \in \mathbb{C}\ \exists i, j \in \mathbb{N}\ .\\ &\qquad \begin{pmatrix} \mathtt{icontains}(b_1, i, j, b_2) \wedge \\ b_1 \in part(\tau_1) \wedge b_2 \in part(\tau_2) \end{pmatrix} \implies \begin{pmatrix} (\tau_1, \tau_2) \in eqs^\sharp \vee \\ (\tau_1, i, j, \tau_2) \in contains^\sharp \end{pmatrix}\end{aligned}$$

Lemma 5. *Given a concrete state $G^\natural = (\epsilon, ptsto^\natural, src, type, part, valid, m, \mathcal{B})$ and a abstract state $G^\sharp = (contains^\sharp, ptsto^\sharp, eqs^\sharp)$ at a program point, we have for two overlapped blocks, if their types are both scalar, then their cell variables must be* *equivalent:*

$$\forall b_1, b_2 \in \mathcal{B}\ \exists \tau_1, \tau_2 \in \mathbb{C}\ .$$
$$\left(\begin{array}{c} b_1 \in part(\tau_1) \wedge b_2 \in part(\tau_2) \wedge \texttt{overlap}(b_1, b_2) \wedge \\ isScalar(type(b_1)) \wedge isScalar(type(b_2)) \end{array} \right) \implies (\tau_1, \tau_2) \in eqs^\sharp$$

As mentioned above, the statement we claim here is that the aliasing between memory blocks in the concrete semantics is captured by the abstract semantics via the equivalence relation between the associated cell variables. There are two kinds of aliasing. The first one is introduced by pointers. Given a memory block, every block it may point during the program execution are *aliased*, Another aliasing is introduced by (partial) overlapping, known as "byte-level aliasing". Two blocks are overlapping if they share a segment of bytes. Only the byte-level aliasing between scalar blocks are taken care of as they are the unit of memory access.

Theorem 2. *Given a program e, let $G_0^\natural \to \cdots \to G_n^\natural$ denote the sequence of state transition of a program execution,*

- *for any block $b \in \mathcal{B}_n$, if there exists $b_1, b_2 \in \mathcal{B}_n$ that $b_1 = ptsto_i^\natural(b)$ and $b_2 = ptsto_j^\natural(b)$, where $0 \leq i, j \leq n$, then $\forall \tau_1, \tau_2 \in \mathbb{C}\ .\ b_1 \in part_n(\tau_1) \wedge b_2 \in part_n(\tau_2) \implies \tau_1 = \tau_2$;*
- *for any two blocks $b_1, b_2 \in \mathcal{B}_n$, if $\texttt{overlap}(b_1, b_2)$ and both $type_i(b_1)$ and $type_i(b_2)$ are scalar where $0 \leq i \leq n$, then $\forall \tau_1, \tau_2 \in \mathbb{C}\ .\ b_1 \in part_i(\tau_1) \wedge b_2 \in part_i(\tau_2) \implies \tau_1 = \tau_2$.*

Proof. The statement holds with lemma 2, 3, 5 and theorem 1. $\square$

3.3.4 Soundness Proof

Proof of Theorem 1

Proof. $G_0^\sharp$ is the initial state. It is obvious that $G_0^\sharp \sqsubseteq^\sharp G$, as $ptsto_0^\sharp$, $contains_0^\sharp$ and $eqs_0^\sharp$ are empty. Then we prove $G_n^\sharp \sqsubseteq^\sharp G$ by induction on the derivation of expression evaluation rules. We assume the pre-state $G^\sharp \sqsubseteq^\sharp G$, then prove that the post-state $G^{\sharp\prime} \sqsubseteq^\sharp G$.

In Const and Var, the post-state is the same as the pre-state and thus the claim holds. For Seq, we have (a) $\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp$ and (b) $\langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp$. By induction on (a), we have $G_1^\sharp \sqsubseteq^\sharp G$; then by induction on (b), we have $G_2^\sharp \sqsubseteq^\sharp G$.

In Addr, by induction on $\langle G^\sharp, e\rangle \Downarrow^\sharp G_1^\sharp$, we have $G_1^\sharp \sqsubseteq^\sharp G$. The abstract store is then updated as $ptsto_{1'}^\sharp = ptsto_1^\sharp[\tau \mapsto \tau']$. Let $G_{1'}^\sharp = (contains_1^\sharp, ptsto_{1'}^\sharp, eqs_1^\sharp)$. According to the constraint generation rule Addr in Fig. 3.6, $G \models \tau \rightharpoonup \tau'$ and thus $\tau^G \overset{G}{\rightharpoonup} \tau'^G$. Therefore, $G_{1'}^\sharp \sqsubseteq^\sharp G$. With lemma 1, $G_2^\sharp = {G_{1'}^\sharp}^+ \sqsubseteq^\sharp G$. Similarly, in Deref and Malloc, we can infer $G_2^\sharp \sqsubseteq^\sharp G$.

In FieldSel, by induction on $\langle G^\sharp, e\rangle \Downarrow^\sharp G_1^\sharp$, we have $G_1^\sharp \sqsubseteq^\sharp G$. The set of contains relation is updated as $contains_{1'}^\sharp = contains_1^\sharp \cup \{(\tau, o, o + |\mathtt{t}|, \tau_f)\}$. Let $G_{1'}^\sharp = (contains_{1'}^\sharp, ptsto_1^\sharp, eqs_1^\sharp)$. We have $G \models \tau \hookrightarrow_{o,o+|\mathtt{t}|} \tau_f$ according to the constraint generation rule FieldSel. Then, $\tau^G \overset{G}{\hookrightarrow}_{o,o+s} \tau_f^G$. Thus $G_{1'}^\sharp \sqsubseteq^\sharp G$. With lemma 1, $G_2^\sharp = {G_{1'}^\sharp}^+ \sqsubseteq^\sharp G$.

In Cast, by induction on $\langle G^\sharp, e\rangle \Downarrow^\sharp G_1^\sharp$, we have $G_1^\sharp \sqsubseteq^\sharp G$. Then in $G_{1'}^\sharp$, the abstract store is updated as $ptsto_{1'}^\sharp = ptsto_1^\sharp[\tau_1 \mapsto \tau_1']$. According to the constraint generation rule Cast, we have $G \models \tau_1 \rightharpoonup \tau_1'$ and thus $\tau_1^G \overset{G}{\rightharpoonup} {\tau_1'}^G$. Also, we have $contains_{1'}^\sharp = contains_1^\sharp \cup cast^\sharp(contains_1^\sharp, \tau_1', \tau_2', |\mathtt{t}|)$. For each $(\tau, i, i + |\mathtt{t}|, \tau_2') \in cast^\sharp(contains_1^\sharp, \tau_1', \tau_2', |\mathtt{t}|)$, with (3.16), we have $(\tau, i, j, \tau_1') \in contains_1^\sharp$

and $\mathsf{source}(\tau)$. By induction (2) on $G_1^\sharp$, we have, $\tau^G \overset{G}{\hookrightarrow}_{i,j} {\tau'_1}^G$ and $\tau^G \in source$. With (3.12), we have $\tau^G \overset{G}{\hookrightarrow}_{i,i+|\mathtt{t}|} {\tau'_2}^G$. Let $G_{1'}^\sharp = (contains_{1'}^\sharp, ptsto_{1'}^\sharp, eqs_1^\sharp)$, then $G_{1'}^\sharp \sqsubseteq^\sharp G$. With lemma 1, $G_2^\sharp = {G_{1'}^\sharp}^+ \sqsubseteq^\sharp G$.

In $\textsc{Assg}_1$, we have (a) $\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp$ and (b) $\langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp$. By induction on (a), we have $G_1^\sharp \sqsubseteq^\sharp G$; then by induction on (b), we have $G_2^\sharp \sqsubseteq^\sharp G$. Then, the set of equivalence relation is updated as $eqs_{2'}^\sharp = eqs_2^\sharp \cup \{(\tau, \tau_2)\}$. According to the constraint generation rule $\textsc{Assg}$, we have $G \models \tau = \tau_2$ and thus $\tau^G = \tau_2^G$. Let $G_{2'}^\sharp = (contains_2^\sharp, ptsto_2^\sharp, eqs_{2'}^\sharp)$, then $G_{2'}^\sharp \sqsubseteq^\sharp G$. With lemma 1, $G_3^\sharp = {G_{2'}^\sharp}^+ \sqsubseteq^\sharp G$.

In $\textsc{Assg}_2$, by induction on $\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp$ and $\langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp$, we have $G_2^\sharp \sqsubseteq^\sharp G$. Then, the set of equivalence relation is updated as $eqs_{2'}^\sharp = eqs_2^\sharp \cup \{(\tau, \tau_2)\}$. According to the constraint generation rule $\textsc{Assg}$, we have $G \models \tau = \tau_2$ and thus $\tau^G = \tau_2^G$. Also, the abstract store is updated as $ptsto_{2'}^\sharp = ptsto_2^\sharp[\tau_1 \mapsto ptsto_2^\sharp(\tau_2), \tau \mapsto ptsto_2^\sharp(\tau_2)]$. With $\tau^G = \tau_2^G$, then $\tau^G \overset{G}{\rightharpoonup} ptsto_2^\sharp(\tau_2)^G$. According to the rule $\textsc{Assg}$, we have $G \models \tau_2 \trianglelefteq \tau_1$. With (3.14), $\tau_1^G \overset{G}{\rightharpoonup} ptsto_2^\sharp(\tau_2)^G$. Let $G_{2'}^\sharp = (contains_2^\sharp, ptsto_{2'}^\sharp, eqs_{2'}^\sharp)$, then $G_{2'}^\sharp \sqsubseteq^\sharp G$. With lemma 1, $G_3^\sharp = {G_{2'}^\sharp}^+ \sqsubseteq^\sharp G$.

In $\textsc{Arith-Op}_1$, by induction on $\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp$ and $\langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp$, we have $G_2^\sharp \sqsubseteq^\sharp G$.

In $\textsc{Arith-Op}_2$, by induction on $\langle G^\sharp, e_1\rangle \Downarrow^\sharp G_1^\sharp$ and $\langle G_1^\sharp, e_2\rangle \Downarrow^\sharp G_2^\sharp$, we have $G_2^\sharp \sqsubseteq^\sharp G$. Then, the abstract store is updated as $ptsto_{2'}^\sharp = ptsto_2^\sharp[\tau \mapsto \tau'_1]$. From $\tau'_1 = ptsto_2^\sharp(\tau_1)$, we have $\tau_1^G \overset{G}{\rightharpoonup} {\tau'_1}^G$. According to the constraint generation rule $\textsc{Arith-Op}$, we have $G \models \tau_1 \trianglelefteq \tau$. With (3.14), we can infer $\tau^G \overset{G}{\rightharpoonup} {\tau'_1}^G$. Then, the set of equivalence relation is updated as $eqs_{2'}^\sharp = eqs_2^\sharp \cup collapse^\sharp(contains_2^\sharp, \tau'_1)$. For each $(\tau'_1, \tau'_c) \in collapse^\sharp(contains_2^\sharp, \tau'_1)$, according to (3.17), there is τ', such that $overlap^G(\tau'_1, \tau')$ and ${\tau'_c}^G \overset{G}{\hookrightarrow}_{k_1,k_2} {\tau'}^G$. From the constraint generation rule $\textsc{Arith-Op}$, we have $G \models \mathsf{collapsed}(\tau)$. With (3.13), we have ${\tau'_1}^G = {\tau'_c}^G$. Let

$G^\sharp_{2'} = (contains^\sharp_2, ptsto^\sharp_{2'}, eqs^\sharp_{2'})$. With lemma 1, $G^\sharp_3 = {G^\sharp_{2'}}^+ \sqsubseteq^\sharp G$. Similarly, for ARITH-OP$_3$, we can infer $G^\sharp_3 \sqsubseteq^\sharp G$.

In ARITH-OP$_4$, by induction on $\langle G^\sharp, e_1\rangle \Downarrow^\sharp G^\sharp_1$ and $\langle G^\sharp_1, e_2\rangle \Downarrow^\sharp G^\sharp_2$, we have $G^\sharp_2 \sqsubseteq^\sharp G$. The abstract store is then updated as $ptsto^\sharp_{2'} = ptsto^\sharp_2[\tau \mapsto \tau'_1]$. From $\tau'_1 = ptsto^\sharp_2(\tau_1)$ and $\tau'_2 = ptsto^\sharp_2(\tau_2)$, we have $\tau_1^G \overset{G}{\rightharpoondown} {\tau'_1}^G$ and $\tau_2^G \overset{G}{\rightharpoondown} {\tau'_2}^G$. According to the constraint generation rule ARITH-OP, we have $G \models \tau_1 \trianglelefteq \tau \wedge \tau_2 \trianglelefteq \tau$. With (3.14), we can infer $\tau^G \overset{G}{\rightharpoondown} {\tau'_1}^G$, $\tau^G \overset{G}{\rightharpoondown} {\tau'_2}^G$ and thus ${\tau'_1}^G = {\tau'_2}^G$. Besides (τ'_1, τ'_2), the set of equivalence relation is updated with two more sets added $collapse^\sharp(contains^\sharp_2, \tau'_1)$ and $collapse^\sharp(contains^\sharp_2, \tau'_2)$. Following the proof of ARITH-OP$_2$ and ARITH-OP$_3$, these newly added equivalence relations are satisfied by G. Let $G^\sharp_{2'} = (contains^\sharp_2, ptsto^\sharp_{2'}, eqs^\sharp_{2'})$. With lemma 1, $G^\sharp_3 = {G^\sharp_{2'}}^+ \sqsubseteq^\sharp G$. $\square$

In order to prove the lemmas in section 3.3.3, we first introduce extra lemmas as follows.

Lemma 6. *Given an expression e and $\langle G^\natural_1, e\rangle \Downarrow^\natural_l \langle loc, G^\natural_2\rangle$, we have*

$$\tau = \Gamma(e) \wedge \mathtt{t} = typeof(\mathcal{T}, e) \implies (loc, |\mathtt{t}|) \in part_2(\tau) \wedge \tau \hookrightarrow_{0,|\mathtt{t}|} \tau$$

Lemma 7. *Given a concrete state $G^\natural = (\epsilon, ptsto^\natural, src, type, part, valid, m, \mathcal{B})$, for any block $b \in \mathcal{B}$, it is contained in its source block: $\forall b \in \mathcal{B} \,.\, \mathtt{contains}(src(b), b)$.*

Lemma 8. *Given a concrete state $G^\natural = (\epsilon, ptsto^\natural, src, type, part, valid, m, \mathcal{B})$, for any source block $b \in \mathtt{range}(src)$, if it shares the same base address as a valid block $b' \in \mathtt{dom}(valid)$, then the associated cell variables of b must be the source cell variables. Furthermore, let $valid(b') = \mathtt{t}$, s be the size of b, and i be the offset of the starting location of b from the base address. If $i \bmod |\mathtt{t}| \neq 0$ or $s \neq |\mathtt{t}|$, then* *the cell size of the associated cell variables of* b *must be* $\top$.

$$\forall b \in \texttt{range}(src)\ \forall \tau \in part(b)\ \exists b' \in \texttt{dom}(valid)\ \exists a \in \mathbb{A}\ \exists i, s_1, s_2 \in \mathbb{N}\ .\\ \begin{pmatrix} b = ((a,i), s_1) \wedge \\ b' = ((a,0), s_2) \wedge \\ valid(b') = \texttt{t} \end{pmatrix} \implies \begin{pmatrix} \mathsf{source}(\tau) \wedge \\ s_1 \neq |\texttt{t}| \vee i \bmod |\texttt{t}| \neq 0 \implies \mathsf{size}(\tau) = \top \end{pmatrix}$$

Lemma 9. *Given a concrete state* $G^\natural = (\epsilon, ptsto^\natural, src, type, part, valid, m, \mathcal{B})$ *and a abstract state* $G^\sharp = (contains^\sharp, ptsto^\sharp, eqs^\sharp)$ *at a program point, for any two source blocks, if their starting locations share the same base address, their cell variables must be equivalent:*

$$\forall b_1, b_2 \in \texttt{range}(src)\ \forall \tau_1, \tau_2 \in \mathbb{C}\ \exists a \in \mathbb{A}\ \exists i_1, i_2, s_1, s_2 \in \mathbb{N}\ .\\ b_1 = ((a, i_1), s_1) \in part(\tau_1) \wedge b_2 = ((a, i_2), s_2) \in part(\tau_2) \implies (\tau_1, \tau_2) \in eqs^\sharp$$

The proof of each lemma is done in two steps. The first step, is to prove the given statement for the initial state $G^\natural_0$. The second step, known as the inductive step, is to prove that, given a state transition $\langle G^\natural, e\rangle \Downarrow^\natural_l \langle loc, G^{\natural\prime}\rangle$, if lemma 2, 3, 4, 5, 6, 7, 8, 9 hold for the pre-state $G^\natural$, the given statement also holds for the post-state $G^{\natural\prime}$.

Proof of Lemma 6.

Proof. This claim can be proved by induction on each rule of the concrete semantics, and the constraint generation rules. □

Proof of Lemma 7.

Proof. In the initial state $G^\natural_0$, $src_0 = \{b_x \mapsto b_x \mid x \in \mathbb{X}\}$. Since each block contains itself, so the lemma holds.

In Const, Seq, Var, Deref, Arith-Op$_1$, Assg$_1$, Assg$_2$, src is not updated and the lemma holds by induction. In Addr, Malloc, Cast$_2$, Arith-Op$_2$, src is updated with fresh blocks that are mapped to itself. Since each block contains itself, so the lemma holds.

In FieldSel, by induction of the lemma on $G_1^\natural$, we know $\texttt{contains}(src_1(b), b)$ and $\texttt{contains}(b, b_f)$, then $\texttt{contains}(src_1(b), b_f)$. So the lemma holds.

In Cast$_1$, $src_2 = src_1[b_l \mapsto b_l, b'_* \mapsto src_1(b_*)]$ and b_l is fresh block. Since $\texttt{contains}(b_l, b_l)$ and $\texttt{contains}(src_1(b_*), b'_*)$, the lemma holds. □

Proof of Lemma 8.

Proof. The lemma holds in the initial state $G_0^\natural$, according to the constraint generation rule Var.

In Const, Seq, Var, Deref, FieldSel, Arith-Op$_1$, Assg$_1$, Assg$_2$, src is not updated. In Addr, Cast$_1$, the newly-added source blocks have no corresponding valid block. So the lemma holds by induction.

In Malloc, we know the newly-added source block b_0 sharing the same base address a_l with the newly-added valid block b, and τ_2 is the cell variable associated with b_0. Considering the offset of the starting location of b_0 from the newly generated base address is 0 and the size of b_0 is $|\texttt{t}|$, then we just need to show

$\mathsf{source}(\tau_2)$, which holds according to the constraint generation rule MALLOC.

In CAST$_2$, $src_2 = \mathtt{ite}(b'_* \in \mathtt{dom}(src_1), src_1[b_l \mapsto b_l], src_1[b_l \mapsto b_l, b'_* \mapsto b'_*])$. b_l is fresh block without corresponding valid block in $valid_1$. If $b'_* \in \mathtt{dom}(src_1)$, the lemma holds by induction.

Otherwise, b'_* is a fresh source block that associated with a single cell variable associated with τ_2 in src_2. By induction of the lemma, we first need to show $\mathsf{source}(\tau_2)$. We know $\neg\mathtt{contains}(src_1(b_*), b'_*)$ and $\mathtt{contains}(src_1(b_*), b_*)$. Let $o \in \mathbb{N}$ be the offset of b'_* from the starting location of $src_1(b_*)$, then $o+|\mathtt{t}| > s_{src}$ where s_{src} is the size of $src_1(b_*)$. By induction on the lemma on $G_1^\natural$, with $\mathtt{isValid}(G_1^\natural, b_*)$, then

$$\forall \tau_{src} \in \mathbb{C} \;.\; src_1(b_*) \in part_1(\tau_{src}) \implies \mathsf{source}(\tau_{src}) \tag{3.18}$$

Together with $\mathsf{cast}(|\mathtt{t}|, \tau_*, \tau_2)$ (from the constraint generation rule CAST), we know

$$\forall \tau_{src} \in \mathbb{C} \;.\; src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} \hookrightarrow_{o,o+|\mathtt{t}|} \tau_2 \tag{3.19}$$

Since $s_{src} \sqsubseteq \mathsf{size}(\tau_{src})$ and $o + |\mathtt{t}| > s_{src}$, according to (3.9), $\mathsf{size}(\tau_{src}) = \top$. (3.19) can be rewritten as

$$\forall \tau_{src} \in \mathbb{C} \;.\; src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} = \tau_2 \wedge \mathsf{size}(\tau_{src}) = \top \wedge \mathsf{size}(\tau_2) = \top \tag{3.20}$$

We can infer $\mathsf{source}(\tau_2)$ from (3.18) and (3.20). Also, we do not need to consider the offset and size of b'_*, since $\mathsf{size}(\tau_2) = \top$. Thus, the lemma holds.

In rule ARITH-OP$_2$, b' is the newly-added source block without corresponding valid block in $valid_2$. b'_* is another newly-added source block if it is fresh. The cell variables associated with b'_* in $part_3$ are $\mathbb{C}_* = \{\tau \mid b_* \in part_2(\tau)\}$. Then we first need to show all the cell variables in $\mathbb{C}_*$ are source cell variables:

$$\forall \tau_* \in \mathbb{C} \; . \; b_* \in part_2(\tau_*) \implies \mathsf{source}(\tau_*) \tag{Arith$_2$-Obj1}$$

Considering $\texttt{isValid}(G_2^\natural, b_*)$, by induction of the lemma on $G_2^\natural$, we can infer

$$\forall \tau_{src} \in \mathbb{C} \; . \; src_2(b_*) \in part_2(\tau_{src}) \implies \mathsf{source}(\tau_{src}) \tag{3.21}$$

By induction of lemma 7 on $G_2^\natural$, $\texttt{contains}(src_2(b_*), b_*)$. According to the relation between $\texttt{contains}$ and $\texttt{icontains}$, we know $\texttt{icontains}(src_2(b_*), i, j, b_*)$ where $i, j \in \mathbb{N}$. By induction of lemma 4 on $G_2^\natural$, we know

$$\forall \tau_{src}, \tau_* \in \mathbb{C} \; . \; \begin{pmatrix} b_* \in part_2(\tau_*) \wedge \\ src_2(b_*) \in part_2(\tau_{src}) \end{pmatrix} \implies \tau_{src} \hookrightarrow_{i,j} \tau_* \vee \tau_{src} = \tau_* \tag{3.22}$$

By induction of lemma 3 on $G_2^\natural$, $\tau_1 \rightharpoonup \tau_*$, where $\tau_1 = \Gamma(e_1)$. With the constraint generation rule Arith-Op, $\mathsf{collapsed}(\tau)$ and $\tau \trianglelefteq \tau_1$. With (3.14), $\tau \trianglelefteq \tau_1 \implies \tau \rightharpoonup \tau_*$. With (3.13), from $\mathsf{collapsed}(\tau)$, (3.22) can be rewritten as

$$\forall \tau_{src}, \tau_* \in \mathbb{C} \; . \; b_* \in part_2(\tau_*) \wedge src_2(b_*) \in part_2(\tau_{src}) \implies \tau_{src} = \tau_* \tag{3.23}$$

From (3.21) and (3.23), (Arith$_2$-Obj1) holds.

Let us consider the offset and size of b'_*. According to the semantics of Arith-Op, $b_* = ((a, i), |\mathtt{t}|)$ and $b'_* = ((a, i \pm v_2), |\mathtt{t}|)$. Considering $\texttt{isValid}(G_2^\natural, b_*)$ and $\texttt{isValid}(G_2^\natural, b'_*)$, there must be $b_{valid} = ((a, 0), s)$ and $valid_2(b_{valid}) = \mathtt{t}'$. Then we just need to show

$$\forall \tau_* \in \mathbb{C} \;.\; b_* \in part_2(\tau_*) \wedge ((i \pm v_2) \bmod |\mathtt{t}'| \neq 0 \vee |\mathtt{t}'| \neq |\mathtt{t}|) \implies \mathsf{size}(\tau_*) = \top \tag{Arith$_2$-Obj2}$$

If $|\mathtt{t}'| \neq |\mathtt{t}|$, from the constraint generation rule Arith-Op, $|\mathtt{t}| \sqsubseteq \mathsf{size}(\tau_*)$. Then, let s_{src} be the size of $src_2(b_*)$. (1) If $s_{src} \neq |\mathtt{t}'|$, by induction of the lemma on $G_2^\natural$, we know $\mathsf{size}(\tau_{src}) = \top$ and thus $\mathsf{size}(\tau_*) = \top$ from (3.23). (2) Otherwise, $s_{src} = |\mathtt{t}'|$. By induction of the lemma 2 on $G_2^\natural$, $|\mathtt{t}'| \sqsubseteq \mathsf{size}(\tau_{src})$. Considering $|\mathtt{t}| \sqsubseteq \mathsf{size}(\tau_*)$, with (3.23), we know $\mathsf{size}(\tau_*) = \top$. Thus (Arith$_2$-Obj2) holds.

Otherwise, $|\mathtt{t}'| = |\mathtt{t}|$. We then consider the other case $(i \pm v_2) \bmod |\mathtt{t}'| \neq 0$. We can infer $(i \bmod |\mathtt{t}|) \neq 0$ from $v_2 \bmod |\mathtt{t}| = 0$. Let s_{src} be the size of $src_2(b_*)$. (1) If $s_{src} = |\mathtt{t}'|$, since the size of b_* is $|\mathtt{t}|$, we can infer $src_2(b_*) = b_*$, considering $\mathtt{contains}(src_2(b_*), b_*)$ by induction of lemma 7 on $G_2^\natural$. Thus, i is also the offset of the starting location of $src_2(b_*)$. Since $i \bmod |\mathtt{t}'| \neq 0$, by induction of the lemma on $G_2^\natural$, $\mathsf{size}(\tau_*) = \top$. (2) Otherwise, $s_{src} \neq |\mathtt{t}'|$. From the above discussion, we can infer $\mathsf{size}(\tau_*) = \mathsf{size}(\tau_{src}) = \top$. Thus, (Arith$_2$-Obj2) holds. □

Proof of Lemma 9.

Proof. In the initial state, the source blocks are all disjoint, so the lemma holds.

In Const, Seq, Var, Deref, FieldSel, Arith-Op$_1$, Assg$_1$, Assg$_2$, *src* is not updated. In Addr, Cast$_1$, the newly-added source blocks are disjoint with each other. So the lemma holds by induction. In Malloc, the new source blocks b_l and b_0 are disjoint with others and thus the lemma holds.

In $\textsc{Cast}_2$, $src_2 = \texttt{ite}(b'_* \in \texttt{dom}(src_1), src_1[b_l \mapsto b_l], src_1[b_l \mapsto b_l, b'_* \mapsto b'_*])$. b_l is fresh block distinct with other blocks. b'_* is added if it is fresh. If b'_* is a fresh source block, we know the cell variable associated with b'_* in $part_2$ is τ_2. So we just need to show $\mathsf{source}(\tau_2)$.

We know $\neg\texttt{contains}(src_1(b_*), b'_*)$ and $\texttt{contains}(src_1(b_*), b_*)$. Let $o \in \mathbb{N}$ be the offset of b'_* from the starting location of $src_1(b_*)$. We know $o + |\mathtt{t}| > s_{src}$, where s_{src} is the size of $src_1(b_*)$. By induction on lemma 8 on $G_1^\natural$, with $\texttt{isValid}(G_1^\natural, b_*)$, we can infer $\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \mathsf{source}(\tau_{src})$. Together with $\mathsf{cast}(|\mathtt{t}|, \tau_*, \tau_2)$ (from the constraint generation rule $\textsc{Cast}$), we can infer

$$\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} \hookrightarrow_{o, o+|\mathtt{t}|} \tau_2 \tag{3.24}$$

Since $s_{src} \sqsubseteq \mathsf{size}(\tau_{src})$ and $o + |\mathtt{t}| > s_{src}$, according to (3.9), $\mathsf{size}(\tau_{src}) = \top$. (3.24) can be rewritten as

$$\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} = \tau_2$$

So the lemma holds.

In $\textsc{Arith-Op}_2$, b' is the newly-added source block and is disjoint with other blocks. b'_* is added if it is fresh. If b'_* is a fresh source block, we know the cell variables associated with b'_* in $part_3$ are $\mathbb{C}_* = \{\tau \mid b_* \in part_2(\tau)\}$. Then we need to show

$$\forall \tau_*, \tau_{src} \in \mathbb{C} \,.\, b_* \in part_2(\tau_*) \wedge src_2(b_*) \in part_2(\tau_{src}) \implies \tau_* = \tau_{src} \tag{Arith$_2$-Obj}$$

By induction of lemma 7 on $G_2^\natural$, $\texttt{contains}(src_2(b_*), b_*)$ and $\texttt{icontains}(src_2(b_*), i, j, b_*)$

where $i, j \in \mathbb{N}$. By induction of lemma 4 on $G_2^\natural$

$$\forall \tau_{src}, \tau_* \in \mathbb{C} \; . \left(\begin{array}{l} b_* \in part_2(\tau_*) \wedge \\ src_2(b_*) \in part_2(\tau_{src}) \end{array} \right) \implies \tau_{src} \hookrightarrow_{i,j} \tau_* \vee \tau_{src} = \tau_* \tag{3.25}$$

By induction on lemma 3 on $G_1^\natural$, $\tau_1 \rightharpoonup \tau_*$ where $\tau_1 = \Gamma(e_1)$. With the constraint generation rule Arith-Op, we have $\mathsf{collapsed}(\tau)$ and $\tau \trianglelefteq \tau_1$. With (3.14), $\tau \trianglelefteq \tau_1 \implies \tau \rightharpoonup \tau_*$. With (3.13), from $\mathsf{collapsed}(\tau)$, (3.25) can be rewritten as

$$\forall \tau_{src}, \tau_* \in \mathbb{C} \; . \; b_* \in part_2(\tau_*) \wedge src_2(b_*) \in part_2(\tau_{src}) \implies \tau_{src} = \tau_*$$

Then (Arith$_2$-Obj) holds. □

Proof of Lemma 2.

Proof. In the initial state $G_0^\natural$, $\mathcal{B}_0 = \{b_x \mid \mathtt{x} \in \mathbb{X}\}$. For each $b_x \in \mathcal{B}_0$, there is only one $\tau_x \in \mathtt{dom}(part_0)$ such that $b_x \in part_0(\tau_x)$, so the lemma obviously holds.

In Const, Var, Deref, Seq, Assg$_1$, Assg$_2$, *part* is not changed and the lemma holds by induction. In Addr and Malloc, *part* is updated in the post-state with newly-added blocks with fresh base address associated with a single cell variable. In other words, these new blocks are associated with a single cell variable in the updated *part* of the post-state. Thus, the lemma holds.

In FieldSel, $part_2$ is updated with block b_f is added to $part_1(\tau_f)$ and the size of b_f is $|\mathtt{t.f}|$. We have $|\mathtt{t.f}| \sqsubseteq \mathsf{size}(\tau_f)$, from the constraint generation rule Dir-Sel. If $b_f \notin \mathcal{B}_1$, then b_f is a fresh block and τ_f is the single cell variable associated with it in $part_2$ of $G_2^\natural$. Both (Eq1) and (Eq2) hold.

Otherwise, $b_f \in \mathcal{B}_1$. By induction of (Eq1) on $part_1$, we have $\forall \tau_f' \in \mathbb{C} \, . \, b_f \in part_1(\tau_f') \implies |\mathtt{t.f}| \sqsubseteq \mathsf{size}(\tau_f')$. Considering $|\mathtt{t.f}| \sqsubseteq \mathsf{size}(\tau_f)$, (Eq1) holds on $part_2$.

In order to prove (Eq2) holds, we need to show

$$\forall \tau_f' \in \mathbb{C} \, . \, b_f \in part_1(\tau_f') \implies \tau_f = \tau_f' \qquad \text{(FieldSel-Obj)}$$

Considering $\mathtt{icontains}(b, o_f, o_f + |\mathtt{t.f}|, b_f)$, where $o_f = \mathit{offset}(\mathtt{t}, \mathtt{f})$ is the offset of b_f from the starting location of b, by induction of lemma 4 on $part_1$, with $b \in part_1(\tau)$, we have

$$\forall \tau_f' \in \mathbb{C} \, . \, b_f \in part_1(\tau_f') \implies \tau \hookrightarrow_{o_f, o_f + |\mathtt{t.f}|} \tau_f' \vee \tau = \tau_f'$$

If $\tau \hookrightarrow_{o_f, o_f + |\mathtt{t.f}|} \tau_f'$, together with $\tau \hookrightarrow_{o_f, o_f + |\mathtt{t.f}|} \tau_f$ (from the constraint generation rule Dir-Sel), we can infer (FieldSel-Obj) holds, with the resolution rule Linear and AntiSym.

If $\tau = \tau_f'$, we have either (1) $\mathsf{size}(\tau) = \top$ or (2) $o_f = 0$, $|\mathtt{t.f}| = |\mathtt{t}|$ and $b = b_f$. If (1) $\mathsf{size}(\tau) = \top$, (FieldSel-Obj) holds according to the resolution rule Scalar and Collapse1. If (2) $b = b_f$, we have $o_f = 0$ and $|\mathtt{t.f}| = |\mathtt{t}|$. $\tau \hookrightarrow_{o_f, o_f + |\mathtt{t.f}|} \tau_f$ can be written as $\tau \hookrightarrow_{0, |\mathtt{t}|} \tau_f$. From $\tau \hookrightarrow_{0, |\mathtt{t}|} \tau$ (with lemma 6), we know $\tau_f = \tau$ and (FieldSel-Obj) holds, according to the resolution rule AntiSym and Linear.

In Cast$_1$, $part_2$ is updated with block b_l and b_*' added, where the size of b_l and b_*' are $|\mathtt{ptr}|$ and $|\mathtt{t}|$. We know $|\mathtt{ptr}| \sqsubseteq \mathsf{size}(\tau_1)$ and $|\mathtt{t}| \sqsubseteq \mathsf{size}(\tau_2)$, according to the constraint generation rule Cast. Considering b_l is a newly generated memory block with a fresh base address, τ_1 is the single cell variable associated with it in

$G_2^\natural$. So the lemma holds on b_l.

The remaining issue is whether b'_* is a fresh block or not. If it is fresh, $b'_* \notin \mathcal{B}_1$, τ_2 is the single cell variable associated with it and the lemma holds. Otherwise, $b'_* \in \mathcal{B}_1$. By induction of (Eq2) on $part_1$, we have $\forall \tau'_2 \in \mathbb{C}\ .\ b'_* \in part_1(\tau'_2) \implies |\mathtt{t}| \sqsubseteq \mathsf{size}(\tau'_2)$. Thus, (Eq2) holds on $part_2$. We only need to show (Eq1) holds, which is equivalent to

$$\forall \tau'_2 \in \mathbb{C}\ .\ b'_* \in part_1(\tau'_2) \implies \tau_2 = \tau'_2 \qquad \text{(Cast}_1\text{-Obj)}$$

By induction of lemma 8 on $G_1^\natural$, with $\mathtt{isValid}(G_1^\natural, b_*)$, we know $\forall \tau \in \mathbb{C}\ .\ src_1(b_*) \in part_1(\tau) \implies \mathsf{source}(\tau)$. With the constraint generation rule Cast,

$$\forall \tau \in \mathbb{C}\ .\ src_1(b_*) \in part_1(\tau) \implies \tau \hookrightarrow_{o,o+|\mathtt{t}|} \tau_2 \qquad (3.26)$$

From $\mathtt{contains}(src_1(b_*), b'_*)$, then $\mathtt{icontains}(src_1(b_*), o, o+|\mathtt{t}|, b'_*)$, where o is also the offset of b'_* from the starting location of $src_1(b_*)$. By induction of lemma 4 on $G_1^\natural$, we have

$$\forall \tau, \tau'_2 \in \mathbb{C}\ .\ src_1(b_*) \in part_1(\tau) \wedge b'_* \in part_1(\tau'_2) \implies \tau \hookrightarrow_{o,o+|\mathtt{t}|} \tau'_2 \vee \tau = \tau'_2$$

If $\tau \hookrightarrow_{o,o+|\mathtt{t}|} \tau'_2$, together with $\tau \hookrightarrow_{o,o+|\mathtt{t}|} \tau_2$, (Cast$_1$-Obj) holds, according to the resolution rule Linear and AntiSym. Otherwise $\tau = \tau'_2$, by induction (Eq2), we have either (1) $\mathsf{size}(\tau) = \top$, then (Cast$_1$-Obj) holds; or (2) $src_1(b_*) = b'_*$, the size of $src_1(b_*)$ is $|\mathtt{t}|$. With $\tau \hookrightarrow_{0,|\mathtt{t}|} \tau$ (by induction of lemma 6), according to (3.26), we have $\tau = \tau_2$ and (Cast$_1$-Obj) holds.

In $\textsc{Cast}_2$, if $b'_* \notin \mathcal{B}_1$, the lemma holds as in $\textsc{Cast}_1$. If $b'_* \in \mathcal{B}_1$, we need to show

$$\forall \tau'_2 \in \mathbb{C} \;.\; b_* \in part_1(\tau'_2) \implies \tau_2 = \tau'_2 \tag{Cast$_2$-Obj}$$

As the proof of $\textsc{Cast}_1$, we have

$$\forall \tau_{src} \in \mathbb{C} \;.\; src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} \hookrightarrow_{o,o+|\mathtt{t}|} \tau_2 \tag{3.27}$$

where o is the offset of b'_* and b_* from the starting location of $src_1(b_*)$. By induction of lemma 7, $\mathtt{contains}(src_1(b_*), b_*)$. Together with $\neg\mathtt{contains}(src_1(b_*), b'_*)$, we can infer $o + |\mathtt{t}| > s_{src}$ where s_{src} is the size of $src_1(b_*)$. Thus, $\mathsf{size}(\tau_{src}) = \top$ from (3.27). Considering $b'_* \in \mathcal{B}_1$, $b'_* \in \mathtt{dom}(src_1)$ and $src_1(b'_*)$ and $src_1(b_*)$ share the same base address. By induction of lemma 9 on $G_1^\natural$, we know

$$\forall \tau_{src}, \tau'_{src} \in \mathbb{C} \;.\; \begin{pmatrix} src_1(b_*) \in part_1(\tau_{src}) \land \\ src_1(b'_*) \in part_1(\tau'_{src}) \end{pmatrix} \implies \tau_{src} = \tau'_{src} \tag{3.28}$$

By induction of lemma 7 on $G_1^\natural$, $\mathtt{contains}(src_1(b'_*), b'_*)$. According to the relation between $\mathtt{contains}$ and $\mathtt{icontains}$, $\mathtt{icontains}(src_1(b'_*), o', o' + |\mathtt{t}|, b'_*)$ where o' is the offset of b'_* from the starting location of $src_1(b'_*)$. By induction of lemma 4 on $G_1^\natural$,

$$\forall \tau'_{src}, \tau'_2 \in \mathbb{C} \;.\; \begin{pmatrix} b'_* \in part_1(\tau'_2) \land \\ src_1(b'_*) \in part_1(\tau'_{src}) \end{pmatrix} \implies \tau'_{src} \hookrightarrow_{o',o'+|\mathtt{t}|} \tau'_2 \lor \tau'_{src} = \tau'_2 \tag{3.29}$$

Since $\mathsf{size}(\tau_{src}) = \top$ and thus $\mathsf{size}(\tau'_{src}) = \top$, we can infer (Cast$_2$-Obj) from (3.27), (3.28) and (3.29), with the resolution rule $\textsc{Size}$ and $\textsc{Scalar}$.

In $\textsc{Arith-Op}_1$, $part_3$ is updated with block b added. First, we know the size of b is $|\texttt{ptr}|$. From the constraint generation rule $\textsc{Cast}$, $|\texttt{ptr}| \sqsubseteq \mathsf{size}(\tau)$. Considering b is a newly generated memory block with a fresh base address, τ is the single cell variable bound with it in $G_3^\natural$. The lemma holds.

In $\textsc{Arith-Op}_2$, $part_3$ is updated with two blocks added b' and b'_*. First, we know the size of b' is $|\texttt{ptr}|$. From the constraint generation rule $\textsc{Cast}$, $|\texttt{ptr}| \sqsubseteq \mathsf{size}(\tau)$. Considering b' is a newly generated memory block with a fresh base address, τ is the single cell variable bound with it in $G_3^\natural$. The lemma holds on b'. The remaining issue is b'_*. We know the size of b'_* and b_* are $|\texttt{t}|$, and the cell variables associated with b'_* in $part_3$ are $\mathbb{C}_* = \{\tau \mid b_* \in part_2(\tau)\}$.

If $b'_* \notin \mathcal{B}_2$, by induction of the lemma on b_*, we have (1) $\forall \tau_1, \tau_2 \in \mathbb{C}_* \,.\, \tau_1 = \tau_2$; (2) $\forall \tau \in \mathbb{C}_* \,.\, |\texttt{t}| \sqsubseteq \mathsf{size}(\tau)$. Thus the lemma holds. Otherwise, $b'_* \in \mathcal{B}_2$, we need to show that

$$\forall \tau_*, \tau'_* \in \mathbb{C} \,.\, b_* \in part_2(\tau_*) \wedge b'_* \in part_2(\tau'_*) \implies \tau_* = \tau'_* \qquad \text{(Arith}_2\text{-Obj)}$$

b_* and b'_* share the same base address, and so are $src_2(b_*)$ and $src_2(b'_*)$. By induction of lemma 9 on $G_2^\natural$,

$$\forall \tau_{src}, \tau'_{src} \in \mathbb{C} \,.\, \begin{pmatrix} src_2(b_*) \in part_2(\tau_{src}) \wedge \\ src_2(b'_*) \in part_2(\tau'_{src}) \end{pmatrix} \implies \tau_{src} = \tau'_{src} \qquad (3.30)$$

By induction of lemma 7 on $G_2^\natural$, $\texttt{contains}(src_2(b_*), b_*)$. According to the relation between $\texttt{contains}$ and $\texttt{icontains}$, $\texttt{icontains}(src_2(b_*), o, o + |\texttt{t}|, b_*)$ where $o \in \mathbb{N}$.

By induction of lemma 4 on $G_2^\natural$, we have

$$\forall \tau_*, \tau_{src} \in \mathbb{C} \, . \begin{pmatrix} b_* \in part_2(\tau_*) \wedge \\ src_2(b_*) \in part_2(\tau_{src}) \end{pmatrix} \implies \tau_{src} \hookrightarrow_{o,o+|\mathtt{t}|} \tau_* \vee \tau_{src} = \tau_* \quad (3.31)$$

Since $b_* = ptsto_2^\natural(b_1)$, according to lemma 6, we have $b_1 \in part_2(\tau_1)$. By induction of lemma 3 on $G_2^\natural$, $\exists \tau_* \in \mathbb{C} \, . \, b_* \in part_2(\tau_*) \implies \tau_1 \rightharpoonup \tau_*$. By induction on (Eq1) on $part_2$, $\forall \tau_* \in \mathbb{C} \, . \, b_* \in part_2(\tau_*) \implies \tau_1 \rightharpoonup \tau_*$ According to the constraint generation rule Arith-Op, we have $\mathsf{collapsed}(\tau)$ and $\tau \trianglelefteq \tau_1$. With (3.14), from $\tau \trianglelefteq \tau_1$, we know $\forall \tau_* \in \mathbb{C} \, . \, b_* \in part_2(\tau_*) \implies \tau \rightharpoonup \tau_*$. With (3.13), from $\mathsf{collapsed}(\tau)$, (3.31) can be rewritten as

$$\forall \tau_{src}, \tau_* \in \mathbb{C} \, . \, src_2(b_*) \in part_2(\tau_{src}) \wedge b_* \in part_2(\tau_*) \implies \tau_{src} = \tau_* \quad (3.32)$$

Since $b'_* \in \mathcal{B}_2$ and thus $b'_* \in \mathtt{dom}(src_2)$, $\mathtt{contains}(src_2(b'_*), b'_*)$ by induction of lemma 7 on $G_2^\natural$. According to the relation between $\mathtt{contains}$ and $\mathtt{icontains}$, $\mathtt{icontains}(src_2(b'_*), o', o' + |\mathtt{t}|, b'_*)$ where $o' \in \mathbb{N}$. By induction of lemma 4 on $G_2^\natural$,

$$\forall \tau'_*, \tau'_{src} \in \mathbb{C} \, . \begin{pmatrix} b'_* \in part_2(\tau'_*) \wedge \\ b'_{src} \in part_2(\tau'_{src}) \end{pmatrix} \implies \tau'_{src} \hookrightarrow_{o',o'+|\mathtt{t}|} \tau'_* \vee \tau'_{src} = \tau'_* \quad (3.33)$$

From (3.33), (1) if $\tau'_{src} = \tau'_*$, then (Arith$_2$-Obj) holds, from (3.30) and (3.32); (2) if $\tau'_{src} \hookrightarrow_{o',o'+|\mathtt{t}|} \tau'_*$, from (3.30) and (3.32), we know

$$\forall \tau_*, \tau'_* \in \mathbb{C} \, . \, b_* \in part_2(\tau_*) \wedge b'_* \in part_2(\tau'_*) \implies \tau_* \hookrightarrow_{o',o'+|\mathtt{t}|} \tau'_*$$

Considering the size of b_* and b'_* are both $|\mathtt{t}|$, by induction on (Eq2), we know

$|\mathsf{t}| \sqsubseteq \mathsf{size}(\tau_*)$ and $|\mathsf{t}| \sqsubseteq \mathsf{size}(\tau'_*)$. With the resolution rule Refl, Linear and AntiSym, we can infer (Arith$_2$-Obj) holds. □

Proof of Lemma 3.

Proof. In the initial state $G_0^\natural$, the points-to store $ptsto_0^\natural$ is empty, so the lemma holds.

In Const, Var, Seq, Arith-Op$_1$, Assg$_1$, FieldSel, $ptsto^\natural$ is not updated and the lemma holds by induction.

In Deref, let $\tau = \Gamma(e)$ and $\tau_* = \Gamma(*e)$, and by induction, we have $b \in part_1(\tau)$ and $b_* \in part_1(\tau_*)$ and $ptsto_1^\natural(b) = b_*$. Considering $ptsto_2^\sharp(\tau) = \tau_*$, the lemma holds.

In Addr, $ptsto_1^\natural = ptsto_0^\natural[b_\& \mapsto b]$. Meanwhile, in the abstract semantics, $ptsto_1^\sharp = ptsto_0^\sharp[\tau \mapsto \tau']$ where $\tau = \Gamma(\&e)$ and $\tau' = \Gamma(e)$. By induction, we know $b \in part_1(\tau')$ and thus $b \in part_2(\tau')$. Also, $b_\& \in part_2(\tau)$. So the lemma holds.

In Cast$_1$, Cast$_2$, $ptsto_2^\natural = ptsto_1^\natural[b_l \mapsto b'_*]$. Meanwhile, in the abstract semantics, $ptsto_2^\sharp = ptsto_1^\sharp[\tau_1 \mapsto \tau_2]$. Consider $b_l \in part_2(\tau_1)$ and $b'_* \in part_2(\tau_2)$, the lemma holds.

In Malloc, $ptsto_2^\natural = ptsto_1^\natural[b_l \mapsto b_0]$. Meanwhile, in the abstract semantics, $ptsto_2^\sharp = ptsto_1^\sharp[\tau_1 \mapsto \tau_2]$. Consider $b_l \in part_2(\tau_1)$ and $b_0 \in part_2(\tau_2)$, the lemma holds.

In Arith-Op_2, $ptsto_3^\natural = ptsto_2^\natural[b' \mapsto b'_*]$. Let $\tau_1 = \Gamma(e_1)$ and thus $\tau_* = ptsto_2^\sharp(\tau_1)$, We know $b' \in part_3(\tau)$ and $b'_* \in part_3(\tau_*)$ where $\tau = \Gamma(e_1 \ \pm \ e_2)$. In the abstract semantics Arith-Op_2 and Arith-Op_4, $ptsto_3^\sharp = ptsto_2^\sharp[\tau \mapsto \tau_*]$. So the lemma holds. By switching the operands, we can infer the lemma holds with abstract semantics Arith-Op_3.

In Assg_2, $ptsto_3^\natural = ptsto_2^\natural[b_1 \mapsto ptsto_2^\natural(b_2)]$. Meanwhile, in the abstract semantics, $ptsto_3^\sharp = ptsto_2^\sharp[\tau_1 \mapsto ptsto_2^\sharp(\tau_2)]$, where $\tau_1 = \Gamma(e_1)$ and $\tau_2 = \Gamma(e_2)$. By induction on $G_2^\natural$ and $G_2^\sharp$, $b_1 \in part_2(\tau_1)$, $b_2 \in part_2(\tau_2)$ and $ptsto_2^\natural(b_2) \in part_2(ptsto_2^\sharp(\tau_2))$. Considering $part_3 = part_2$, the lemma holds. $\square$

Proof of Lemma 4.

Proof. In the initial state $G_0^\natural$, the memory blocks in $\mathcal{B}_0$ are disjoint, so the lemma holds.

In Const, Var, Deref, Seq, Arith-Op_1, Assg_1, Assg_2, $\mathcal{B}$ is not updated and the lemma holds by induction. In Addr, Malloc, the blocks are newly generated with fresh base address. They are disjoint with existing memory blocks an the lemma holds.

In FieldSel, $\mathcal{B}_2 = \mathcal{B}_1 \cup \{b_f\}$. If $b_f \in \mathcal{B}_1$, the lemma holds by induction. Otherwise, $b_f \notin \mathcal{B}_1$ and τ_f is the single cell variable associated with b_f in $part_2$. In this case, we need to show

$$\forall b_1 \in \mathcal{B}_1\, \forall \tau_1 \in \mathbb{C}\, \exists i_1, j_1 \in \mathbb{N}\,. \qquad \text{(FieldSel-Obj1)}$$
$$b_1 \in part_1(\tau_1) \land \texttt{icontains}(b_1, i_1, j_1, b_f) \implies \tau_1 \hookrightarrow_{i_1,j_1} \tau_f \lor \tau_1 = \tau_f$$
$$\forall b_2 \in \mathcal{B}_1\, \forall \tau_2 \in \mathbb{C}\, \exists i_2, j_2 \in \mathbb{N}\,. \qquad \text{(FieldSel-Obj2)}$$
$$b_2 \in part_1(\tau_2) \land \texttt{icontains}(b_f, i_2, j_2, b_2) \implies \tau_f \hookrightarrow_{i_2,j_2} \tau_2 \lor \tau_2 = \tau_f$$

By induction on lemma 7 on $G_1^\natural$, $\texttt{contains}(src_1(b), b)$. According to the relation between $\texttt{contains}$ and $\texttt{icontains}$, $\texttt{icontains}(src_1(b), i_3, j_3, b)$ where $i_3, j_3 \in \mathbb{N}$ and $j_3 = i_3 + |\texttt{t}|$. By induction of lemma 6 on $G_1^\natural$, $b \in part_1(\tau)$. By induction on this lemma on $G_1^\natural$,

$$\forall \tau_{src} \in \mathbb{C}\,.\ src_1(b) \in part_1(\tau_{src}) \implies \tau_{src} \hookrightarrow_{i_3,j_3} \tau \lor \tau_{src} = \tau \tag{3.34}$$

According to the constraint generation rule Dir-Sel, $\tau \hookrightarrow_{o_f, o_f+|\texttt{t.f}|} \tau_f$ where $o_f = \mathit{offset}(\texttt{t}, \texttt{f})$. With resolution rule Trans, from (3.34)

$$\forall \tau_{src} \in \mathbb{C}\,.\ src_1(b) \in part_1(\tau_{src}) \implies \tau_{src} \hookrightarrow_{i_3+o_f, i_3+o_f+|\texttt{t,f}|} \tau_f \lor \tau_{src} = \tau_f \tag{3.35}$$

Let us first consider (FieldSel-Obj1). For block $b_1 \in \mathcal{B}_1$ that $\texttt{contains}(b_1, b_f)$, we know either (1) $\texttt{contains}(src_1(b), b_1)$ (b_1 is within the source block of b); (2) $\neg\texttt{contains}(src_1(b), b_1)$ (b_1 is not within the source block of b).

From (1) $\texttt{contains}(src_1(b), b_1)$, then $\texttt{icontains}(src_1(b), i_4, j_4, b_1)$ where $i_4, j_4 \in$

$\mathbb{N}$. By induction of the lemma,

$$\forall \tau_{src}, \tau_1 \in \mathbb{C} \ . \left(\begin{array}{l} b_1 \in part_1(\tau_1) \land \\ src_1(b) \in part_1(\tau_{src}) \end{array} \right) \implies \tau_{src} \hookrightarrow_{i_4,j_4} \tau_1 \lor \tau_{src} = \tau_1 \qquad (3.36)$$

From $\texttt{icontains}(src_1(b), i_4, j_4, b_1)$ and $\texttt{icontains}(src_1(b), i_3 + o_f, i_3 + o_f + |\texttt{t.f}|, b_f)$, with $\texttt{contains}(b_1, b_f)$, we know $i_4 \leq i_3 + o_f < i_3 + o_f + |\texttt{t.f}| \leq j_4$. Let $k_1 = i_3 + o_f - i_4$ and $k_2 = i_3 + o_f + |\texttt{t.f}| - i_4$. According to the resolution rule LINEAR, with (3.35) and (3.36), we know $\tau_1 \hookrightarrow_{k_1,k_2} \tau_f \lor \tau_1 = \tau_f$. Thus (FieldSel-Obj1) holds.

From (2) $\neg\texttt{contains}(src_1(b), b_1)$, we know both b_1 and b contains b_f, then $\texttt{overlap}(src_1(b), b_1)$ and thus $\texttt{overlap}(src_1(b), src_1(b_1))$. By induction of lemma 9 on $G_1^\natural$

$$\forall \tau_{src}, \tau'_{src} \in \mathbb{C} \ . \ src_1(b) \in part_1(\tau_{src}) \land src_1(b_1) \in part_1(\tau'_{src}) \implies \tau_{src} = \tau'_{src}$$

Let b_{valid} is the valid block of $src_1(b)$ and $src_1(b_1)$, where $valid_1(b_{valid}) = \texttt{t}'$. Let s_1 and s_2 are the size of $src_1(b)$ and $src_1(b_1)$. Let o_1 and o_2 are the offset of the starting location of $src_1(b)$ and $src_1(b_1)$ from the starting location of b_{valid}. From $\texttt{overlap}(src_1(b), src_1(b_1))$, we can infer (1) $s_1 \neq |\texttt{t}'|$, or (2) $s_2 \neq |\texttt{t}'|$, or (3) o_1 mod $|\texttt{t}'| \neq 0$, or (4) o_2 mod $|\texttt{t}'| \neq 0$. In either case, by induction of lemma 8, we can infer $\mathsf{size}(\tau'_{src}) = \mathsf{size}(\tau_{src}) = \top$. With the resolution rule SCALAR and COLLAPSE1, from (3.35) and (3.36), $\tau_{src} = \tau_1$, $\tau_{src} = \tau_f$ and thus $\tau_1 = \tau_f$. Therefore, (FieldSel-Obj1) also holds.

Let us then consider (FieldSel-Obj2). For block $b_2 \in \mathcal{B}$ that $\texttt{contains}(b_f, b_2)$, we know $\texttt{contains}(src_1(b), b_2)$ and $\texttt{icontains}(src_1(b), i_5, j_5, b_2)$ where $i_5, j_5 \in \mathbb{N}$.

By induction of the lemma

$$\forall \tau_{src}, \tau_2 \in \mathbb{C} \;.\; \begin{pmatrix} b_2 \in part_1(\tau_2) \wedge \\ src_1(b) \in part_1(\tau_{src}) \end{pmatrix} \implies \tau_{src} \hookrightarrow_{i_5, j_5} \tau_2 \vee \tau_{src} = \tau_2 \tag{3.37}$$

Considering $\texttt{icontains}(src_1(b), i_3 + o_f, i_3 + o_f + |\texttt{t.f}|, b_f)$, and $\texttt{contains}(b_f, b_2)$, we know $i_3 + o_f \leq i_5 < j_5 \leq i_3 + o_f + |\texttt{t.f}|$. Let $k_1 = i_5 - i_3 - o_f$ and $k_2 = j_5 - i_3 - o_f$. From (3.35) and (3.37), from the resolution rule LINEAR, we know $\tau_2 \hookrightarrow_{k_1, k_2} \tau_f \vee \tau_2 = \tau_f$, and thus (FieldSel-Obj2) holds.

In CAST$_1$, $\mathcal{B}_2$ has two blocks added b_l and b'_*. b_l is disjoint with other blocks, as its base address is fresh. If $b'_* \in \mathcal{B}_1$, the lemma holds by induction. Otherwise, $b'_* \notin \mathcal{B}_1$ and τ_2 is the only cell variable associated with it in the updated $part_2$. We need to show

$$\forall b_1 \in \mathcal{B}_1 \, \forall \tau \in \mathbb{C} \, \exists i_1, j_1 \in \mathbb{N} \;. \tag{Cast$_1$-Obj1}$$
$$b_1 \in part_1(\tau) \wedge \texttt{icontains}(b_1, i_1, j_1, b'_*) \implies \tau \hookrightarrow_{i_1, j_1} \tau_2 \vee \tau = \tau_2$$
$$\forall b_2 \in \mathcal{B}_1 \, \forall \tau \in \mathbb{C} \, \exists i_2, j_2 \in \mathbb{N} \;. \tag{Cast$_1$-Obj2}$$
$$b_2 \in part_1(\tau) \wedge \texttt{icontains}(b'_*, i_2, j_2, b_2) \implies \tau_2 \hookrightarrow_{i_2, j_2} \tau \vee \tau = \tau_2$$

By induction of lemma 7 on $G_1^\natural$, $\texttt{contains}(src_1(b_*), b_*)$. Let $o \in \mathbb{N}$ is the offset of b_* from the starting location of $src_1(b_*)$, then $\texttt{icontains}(src_1(b_*), o, o + s, b_*)$. By induction of the lemma

$$\forall \tau_{src}, \tau_* \in \mathbb{C} \;.\; \begin{pmatrix} b_* \in part_1(\tau_*) \wedge \\ src_1(b_*) \in part_1(\tau_{src}) \end{pmatrix} \implies \tau_{src} \hookrightarrow_{o, o+s} \tau_* \vee \tau_{src} = \tau_*$$

Since $\texttt{isValid}(G_1^\natural, b_*)$, with lemma 8, we know $\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \mathsf{source}(\tau_{src})$. From the constraint generation rule CAST, $\mathsf{cast}(|\mathtt{t}|, \tau_*, \tau_2)$,

$$\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} \hookrightarrow_{o,o+|\mathtt{t}|} \tau_2$$

Let us first consider the block $b_1 \in \mathcal{B}_1$ such that $\texttt{contains}(b_1, b'_*)$, we know $\texttt{contains}(src_1(b_*), b_1)$, or $\neg\texttt{contains}(src_1(b_*), b_1)$. Let us then consider the block $b_2 \in \mathcal{B}_1$ that $\texttt{contains}(b'_*, b_2)$, we know $\texttt{contains}(src_1(b_*), b_2)$. Similar as the proof of FIELDSEL, both (Cast_1-Obj1) and (Cast_1-Obj2) hold.

In CAST_2, we know $\neg\texttt{contains}(src_1(b_*), b'_*)$ and $\texttt{contains}(src_1(b_*), b_*)$. Let $o \in \mathbb{N}$ be the offset of b_* and b'_* from the starting location of $src_1(b_*)$. We know $o + |\mathtt{t}| > s_{src}$, where s_{src} is the size of $src_1(b_*)$. By induction of lemma 8 on $G_1^\natural$, with $\texttt{isValid}(G_1^\natural, b_*)$, we can infer $\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \mathsf{source}(\tau_{src})$. Together with $\mathsf{cast}(|\mathtt{t}|, \tau_*, \tau_2)$, as in CAST_1, we can infer

$$\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} \hookrightarrow_{o,o+|\mathtt{t}|} \tau_2 \tag{3.38}$$

Since $s_{src} \sqsubseteq \mathsf{size}(\tau_{src})$ and $o + |\mathtt{t}| > s_{src}$, according to (3.9), $\mathsf{size}(\tau_{src}) = \top$. (3.38) can be rewritten as

$$\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} = \tau_2 \tag{3.39}$$

For any block $b \in \mathcal{B}_1$ that either $\texttt{contains}(b'_*, b)$ or $\texttt{contains}(b, b'_*)$, we know

$src_1(b)$ and $src_1(b_*)$ share the same base address. By induction of lemma 9 on $G_1^\natural$

$$\forall \tau_{src}, \tau'_{src} \in \mathbb{C} \;.\; src_1(b) \in part_1(\tau'_{src}) \wedge src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} = \tau'_{src} \tag{3.40}$$

By induction of lemma 7 on $G_1^\natural$, $\texttt{contains}(src_1(b), b)$. Considering $\mathsf{size}(\tau_{src}) = \top$, by induction of the lemma,

$$\forall \tau, \tau'_{src} \in \mathbb{C} \;.\; b \in part_1(\tau) \wedge src_1(b) \in part_1(\tau'_{src}) \implies \tau'_{src} = \tau \tag{3.41}$$

From (3.39), (3.40) and (3.41), the lemma holds.

In Arith-Op$_2$, $\mathcal{B}_3 = \mathcal{B}_2 \cup \{b', b'_*\}$. b' is disjoint with other blocks, as its base address is fresh. If $b'_* \in \mathcal{B}_2$, the lemma holds by induction. Otherwise, $b'_* \notin \mathcal{B}_2$. We know the cell variables associated with b'_* in $part_3$ are $\mathbb{C}_* = \{\tau \mid b_* \in part_2(\tau)\}$. Then we need to show

$$\forall b_1 \in \mathcal{B}_2 \, \forall \tau_1, \tau_* \in \mathbb{C} \, \exists i_1, j_1 \in \mathbb{N} \;.$$
$$\begin{pmatrix} b_1 \in part_2(\tau_1) \wedge \\ b_* \in part_2(\tau_*) \end{pmatrix} \wedge \texttt{icontains}(b_1, i_1, j_1, b'_*) \implies \tau_1 \hookrightarrow_{i_1, j_1} \tau_* \vee \tau_1 = \tau_* \tag{Arith$_2$-Obj1}$$

$$\forall b_2 \in \mathcal{B}_2 \, \forall \tau_2, \tau_* \in \mathbb{C} \, \exists i_2, j_2 \in \mathbb{N} \;.$$
$$\begin{pmatrix} b_2 \in part_2(\tau_2) \wedge \\ b_* \in part_2(\tau_*) \end{pmatrix} \wedge \texttt{icontains}(b'_*, i_2, j_2, b_2) \implies \tau_* \hookrightarrow_{i_2, j_2} \tau_2 \vee \tau_2 = \tau_* \tag{Arith$_2$-Obj2}$$

By induction of lemma 7 on $G_2^\natural$, $\texttt{contains}(src_2(b_*), b_*)$. According to the relation between $\texttt{contains}$ and $\texttt{icontains}$, $\texttt{icontains}(src_2(b_*), i_3, j_3, b_*)$ where

$i_3, j_3 \in \mathbb{N}$. By induction of the lemma

$$\forall \tau_{src}, \tau_* \in \mathbb{C}\ .\ \begin{pmatrix} b_* \in part_2(\tau_*) \land \\ src_2(b_*) \in part_2(\tau_{src}) \end{pmatrix} \implies \tau_{src} \hookrightarrow_{i_3,j_3} \tau_* \lor \tau_{src} = \tau_* \quad (3.42)$$

By induction of lemma 3 on $G_2^\natural$, $\tau_1 \rightharpoonup \tau_*$ where $\tau_1 = \Gamma(e_1)$. With the constraint generation rule Arith-Op, we have $\mathsf{collapsed}(\tau)$ and $\tau \trianglelefteq \tau_1$. With (3.14), $\tau \trianglelefteq \tau_1 \implies \tau \rightharpoonup \tau_*$. With (3.13), from $\mathsf{collapsed}(\tau)$, (3.42) can be rewritten as

$$\forall \tau_{src}, \tau_* \in \mathbb{C}\ .\ b_* \in part_2(\tau_*) \land src_2(b_*) \in part_2(\tau_{src}) \implies \tau_{src} = \tau_* \quad (3.43)$$

Let us first consider (Arith$_2$-Obj1). For the block $b_1 \in \mathcal{B}_2$ that $\mathtt{contains}(b_1, b'_*)$, we know $src_2(b_1)$ and $src_2(b_*)$ share the same base address. By induction of lemma 9 on $G_2^\natural$

$$\forall \tau_{src}, \tau'_{src} \in \mathbb{C}\ .\ \begin{pmatrix} src_2(b_*) \in part_2(\tau_{src}) \land \\ src_2(b_1) \in part_2(\tau'_{src}) \end{pmatrix} \implies \tau_{src} = \tau'_{src} \quad (3.44)$$

By induction of lemma 7 on $G_2^\natural$, $\mathtt{contains}(src_2(b_1), b_1)$. According to the relation between $\mathtt{contains}$ and $\mathtt{icontains}$, $\mathtt{icontains}(src_2(b_1), i_4, j_4, b_1)$ where $i_4, j_4 \in \mathbb{N}$. By induction of the lemma

$$\forall \tau_1, \tau'_{src} \in \mathbb{C}\ .\ \begin{pmatrix} b_1 \in part_2(\tau_1) \land \\ src_2(b_1) \in part_2(\tau'_{src}) \end{pmatrix} \implies \tau'_{src} \hookrightarrow_{i_4,j_4} \tau_1 \lor \tau_1 = \tau'_{src} \quad (3.45)$$

From (3.43), (3.44) and (3.45), we know

$$\forall \tau_1, \tau_* \in \mathbb{C} \;.\; b_1 \in part_2(\tau_1) \wedge b_* \in part_2(\tau_*) \implies \tau_* \hookrightarrow_{i_4,j_4} \tau_1 \vee \tau_1 = \tau_*$$

If $\tau_1 = \tau_*$, then (Arith$_2$-Obj1) holds. If $\tau_* \hookrightarrow_{i_4,j_4} \tau_1$, considering $\texttt{contains}(b_1, b'_*)$, we have either (1) $\textsf{size}(\tau_*) = \top$ or (2) $b'_* = b_1$, $i_4 = 0$ and $j_4 = |\texttt{t}|$. In either case, $\tau_* = \tau_1$ and (Arith$_2$-Obj1) holds.

Let us consider (Arith$_2$-Obj2) then. For block $b_2 \in \mathcal{B}_2$ that $\texttt{contains}(b'_*, b_2)$, $\texttt{icontains}(b'_*, i_2, j_2, b_2)$ where $i_2, j_2 \in \mathbb{N}$.

From $b_2 \in \mathcal{B}_2$, we know $b_2 \in src_2(b_2)$ and $\texttt{contains}(src_2(b_2), b_2)$, by induction of lemma 7 on $G_2^\natural$. According to the relation between $\texttt{contains}$ and $\texttt{icontains}$, $\texttt{icontains}(src_2(b_2), i_5, j_5, b_2)$ where $i_5, j_5 \in \mathbb{N}$. By induction of lemma 4 on $G_2^\natural$,

$$\forall \tau_2, \tau''_{src} \in \mathbb{C} \;.\; \left(\begin{array}{l} b_2 \in part_2(\tau_2) \wedge \\ src_2(b_2) \in part_2(\tau''_{src}) \end{array} \right) \implies \tau''_{src} \hookrightarrow_{i_5,j_5} \tau_2 \vee \tau_2 = \tau''_{src} \quad (3.46)$$

By induction of lemma 9 on $G_2^\natural$,

$$\forall \tau_{src}, \tau''_{src} \in \mathbb{C} \;.\; src_2(b_*) \in part_2(\tau_{src}) \wedge src_2(b_2) \in part_2(\tau''_{src}) \implies \tau_{src} = \tau''_{src} \quad (3.47)$$

From (3.43) and (3.47), we know $\tau''_{src} = \tau_*$. (3.46) can be rewritten as

$$\forall \tau_2, \tau_* \in \mathbb{C} \;.\; b_2 \in part_2(\tau_2) \wedge b_* \in part_2(\tau_*) \implies \tau_* \hookrightarrow_{i_5,j_5} \tau_2 \vee \tau_2 = \tau_* \quad (3.48)$$

Let s_{src}, s''_{src} are the size of $src_2(b_*)$ and $src_2(b_2)$. Let b_{valid} is the valid block of $src_2(b_*)$ and $src_2(b_2)$ where $valid_2(b_{valid}) = \texttt{t}'$. Then we split the proof into following cases.

(1) $s_{src} \neq |\mathtt{t}'|$ or $s''_{src} \neq |\mathtt{t}'|$, by induction of lemma 8 on $G_2^\natural$, according to (3.47), we can infer $\mathsf{size}(\tau_{src}) = \mathsf{size}(\tau''_{src}) = \top$. With the resolution rule SCALAR and COLLAPSE1, we have infer following formula and thus (Arith$_2$-Obj2) holds.

$$\forall \tau_2, \tau_* \in \mathbb{C} \;.\; b_2 \in part_2(\tau_2) \wedge b_* \in part_2(\tau_*) \implies \tau_2 = \tau_*$$

(2) $|\mathtt{t}'| \neq |\mathtt{t}|$. With $|\mathtt{t}| \sqsubseteq \mathsf{size}(\tau_*)$ (from the constraint generation rule ARITH-OP) and $|\mathtt{t}'| \sqsubseteq \mathsf{size}(\tau_{src})$ (by induction of the lemma 2 on $G_2^\natural$), since $\tau_{src} = \tau_*$, we know $\mathsf{size}(\tau_{src}) = \top$. Similarly, we can infer (Arith$_2$-Obj2) holds.

(3) $|\mathtt{t}'| = |\mathtt{t}| = s_{src} = s''_{src}$. Since $\mathtt{contains}(src_2(b_*), b_*)$, and the size of $src_2(b_*)$ is the same as the size of b_*, we then can first infer $src_2(b_*) = b_*$.

Moreover, from $\mathtt{icontains}(b'_*, i_2, j_2, b_2)$ and $\mathtt{icontains}(src_2(b_2), i_5, j_5, b_2)$ where $i_2, j_2, i_5, j_5 \in \mathbb{N}$, we can infer either $b'_* = src_2(b_2)$, or they are overlapping. If $b'_* = src_2(b_2)$, then $i_2 = i_5$, $j_2 = j_5$, and (Arith$_2$-Obj2) holds according to (3.48).

Otherwise, let o''_{src} and o' are the offset of the starting location of $src_2(b_2)$ and b'_* from the starting location of b_{valid}, then

$$o''_{src} < o' < o''_{src} + |\mathtt{t}| \vee o' < o''_{src} < o' + |\mathtt{t}| \tag{3.49}$$

since b'_* and $src_2(b_2)$ are overlapping. From (3.49), we know either (3.1) o''_{src} mod $|\mathtt{t}| \neq 0$ or (3.2) o' mod $|\mathtt{t}| \neq 0$. For (3.1), by induction of lemma 8, $\mathsf{size}(\tau''_{src}) = \top$. For (3.2), we have $o' = i \pm v_2$ where i is the offset of b_* and $src_2(b_*)$ (since $src_2(b_*) = b_*$). Considering v_2 mod $|\mathtt{t}| = 0$, then i mod $|\mathtt{t}| \neq 0$. By induction of lemma 8, $\mathsf{size}(\tau_{src}) = \top$. From the above case, (Arith$_2$-Obj2) holds.

□

Proof of Lemma 5.

Proof. Before getting to the proof, we first claim that in a concrete state $G^\natural = (\epsilon, ptsto^\natural, src, type, part, valid, m, \mathcal{B})$, any block $b \in \mathcal{B}$, if $type(b)$ is scalar, then all the cell variables associated with b is scalar:

$$\forall b \in \mathcal{B} \, \forall \tau \in \mathbb{C} \, . \, b \in part(\tau) \wedge isScalar(type(b)) \implies \mathsf{scalar}(\tau) \tag{3.50}$$

The claim obviously holds according the semantics rules and the constraints generation rules.

In the initial state $G_0^\natural$, the memory blocks in $\mathcal{B}_0$ are disjoint, so the lemma holds.

In Const, Var, Deref, Seq, Assg_1, Assg_2, $\mathcal{B}$ is not updated and the lemma holds by induction. In Addr, Malloc, Arith-Op_1, the blocks are newly generated with fresh base address. They are disjoint with existing memory blocks.

In FieldSel, $\mathcal{B}_2 = \mathcal{B}_1 \cup \{b_f\}$. If `t.f` is with non-scalar type, then the lemma holds by induction. If $b_f \in \mathcal{B}_1$, the lemma holds by induction. Otherwise `t.f` is scalar, from the constraint generation rule Dir-Sel, we know $\mathsf{scalar}(\tau_f)$ and $\tau \hookrightarrow_{o_f, o_f + |\mathtt{t.f}|} \tau_f$ where $o_f = \mathit{offset}(\mathtt{t}, \mathtt{f})$. Considering b_f is a new scalar block and associated with a single cell variable τ_f, we just need to show

$$\begin{aligned} &\forall b' \in \mathcal{B}_1 \, \forall \tau' \in \mathbb{C} \, . \\ &\qquad b' \in part_1(\tau') \wedge isScalar(type_1(b')) \wedge \mathtt{overlap}(b', b_f) \implies \tau_f = \tau' \end{aligned} \tag{FieldSel-Obj}$$

From $isScalar(type_1(b'))$, with (3.50), $\forall \tau' \in \mathbb{C} \, . \, b' \in part_1(\tau') \implies \mathsf{scalar}(\tau')$. Considering $\texttt{contains}(b, b_f)$, from $\texttt{overlap}(b', b_f)$, we can infer either (1) $src_1(b') = src_1(b)$, or (2) $\texttt{overlap}(src_1(b'), src_1(b))$.

If (1) $src_1(b') = src_1(b)$, by induction on lemma 7 on $G_1^\natural$, $\texttt{contains}(src_1(b), b)$ and $\texttt{contains}(src_1(b), b')$. From $\texttt{contains}(src_1(b), b)$, $\texttt{icontains}(src_1(b), i_1, j_1, b)$ where $i_1, j_1 \in \mathbb{N}$. By induction of lemma 4 on $G_1^\natural$

$$\forall \tau_{src} \in \mathbb{C} \;.\; src_1(b) \in part_1(\tau_{src}) \implies \tau_{src} = \tau \vee \tau_{src} \hookrightarrow_{i_1, j_1} \tau \tag{3.51}$$

With $\tau \hookrightarrow_{o_f, o_f + |\texttt{t.f}|} \tau_f$, according to the resolution rule $\textsc{Trans}$, from (3.51)

$$\forall \tau_{src} \in \mathbb{C} \;.\; src_1(b) \in part_1(\tau_{src}) \implies \tau_{src} = \tau_f \vee \tau_{src} \hookrightarrow_{i_1 + o_f, i_1 + o_f + |\texttt{t.f}|} \tau_f \tag{3.52}$$

From $\texttt{contains}(src_1(b), b')$, we know $\texttt{icontains}(src_1(b), i_2, j_2, b')$ where $i_2, j_2 \in \mathbb{N}$. By induction of lemma 4 on $G_1^\natural$

$$\forall \tau_{src}, \tau' \in \mathbb{C} \;.\; \begin{pmatrix} b' \in part_1(\tau') \wedge \\ src_1(b) \in part_1(\tau_{src}) \end{pmatrix} \implies \tau_{src} = \tau' \vee \tau_{src} \hookrightarrow_{i_2, j_2} \tau' \tag{3.53}$$

We know $\texttt{icontains}(src_1(b), i_1 + o_f, i_1 + o_f + |\texttt{t.f}|, b_f)$, from $\texttt{icontains}(b, o_f, o_f + |\texttt{t.f}, b_f)$ and $\texttt{icontains}(src_1(b), i_1, j_1, b)$, Considering $\texttt{overlap}(b', b_f)$, we know $i_1 + o_f \leq i_2 < i_1 + o_f + |\texttt{t.f}| \vee i_2 \leq i_1 + o_f < j_2$. From $\mathsf{scalar}(\tau_f)$ and $\forall \tau' \in \mathbb{C} \;.\; b' \in part_1(\tau') \implies \mathsf{scalar}(\tau')$, according to the resolution rule $\textsc{Overlap}$, $\forall \tau' \in \mathbb{C} \;.\; b' \in part_1(\tau') \implies \tau_f = \tau'$ and thus (FieldSel-Obj) holds.

If (2) $\texttt{overlap}(src_1(b), src_1(b'))$, by induction of lemma 9 on $G_1^\natural$

$$\forall \tau_{src}, \tau'_{src} \in \mathbb{C} \;.\; src_1(b) \in part_1(\tau_{src}) \wedge src_1(b') \in part_1(\tau'_{src}) \implies \tau_{src} = \tau'_{src} \tag{3.54}$$

Let b_{valid} is the valid block of $src_1(b)$ and $src_1(b')$, where $valid_1(b_{valid}) = \texttt{t}'$. Let s_1 and s_2 are the size of $src_1(b)$ and $src_1(b')$. Let o_1 and o_2 are the offset of the starting location of $src_1(b)$ and $src_1(b')$ from the starting location of b_{valid}. From $\texttt{overlap}(src_1(b), src_1(b'))$, we can infer (1) $s_1 \neq |\texttt{t}'|$, or (2) $s_2 \neq |\texttt{t}'|$, or (3) $o_1 \bmod |\texttt{t}'| \neq 0$, or (4) $o_2 \bmod |\texttt{t}'| \neq 0$. In either case, by induction of lemma 8, we can infer $\mathsf{size}(\tau'_{src}) = \mathsf{size}(\tau_{src}) = \top$.

With $\texttt{contains}(src_1(b), b_f)$ and $\texttt{icontains}(src_1(b), i_3, j_3, b_f)$ where $i_3, j_3 \in \mathbb{N}$, By induction of lemma 7 on $G_1^\natural$, according to the resolution rule SCALAR and COLLAPSE1, from $\mathsf{size}(\tau_{src}) = \top$,

$$\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b) \in part_1(\tau_{src}) \implies \tau_{src} = \tau_f \tag{3.55}$$

With $\texttt{contains}(src_1(b'), b')$ and $\texttt{icontains}(src_1(b'), i_4, j_4, b')$ where $i_4, j_4 \in \mathbb{N}$. Similarly, we can infer

$$\forall \tau'_{src}, \tau' \in \mathbb{C} \,.\, src_1(b') \in part_1(\tau'_{src}) \wedge b' \in part_1(\tau') \implies \tau'_{src} = \tau' \tag{3.56}$$

From (3.54), (3.55) and (3.56), we can infer (FieldSel-Obj) holds.

In CAST_1 and CAST_2, $\mathcal{B}_2 = \mathcal{B}_1 \cup \{b_l, b'_*\}$. b_l is disjoint with other blocks. If $b'_* \in \mathcal{B}_1$ or its type $\texttt{t}$ is not scalar, the lemma holds. Otherwise, $\texttt{t}$ is scalar, from the constraint generation rule CAST, we know $\mathsf{scalar}(\tau_2)$. Considering b'_* is a new scalar block and associated with a single cell variable τ_2, we just need to show

$$\forall b \in \mathcal{B}_1 \, \forall \tau \in \mathbb{C} \,.\, b \in part_1(\tau) \wedge isScalar(type_1(b)) \wedge \texttt{overlap}(b, b'_*) \implies \tau = \tau_2 \tag{Cast-Obj}$$

From $isScalar(type_1(b))$, with (3.50), $\forall \tau \in \mathbb{C} \,.\, b \in part_1(\tau) \implies \mathsf{scalar}(\tau)$.

In $\textsc{Cast}_1$, we know $\mathtt{contains}(src_1(b_*), b'_*)$ and $\mathtt{overlap}(b, b'_*)$, then we have $src_1(b) = src_1(b_*) \vee \mathtt{overlap}(src_1(b), src_1(b_*))$. Similar as $\textsc{FieldSel}$, we can infer (Cast-Obj) holds.

In $\textsc{Cast}_2$, we know $\neg\mathtt{contains}(src_1(b_*), b'_*)$ and $\mathtt{contains}(src_1(b_*), b_*)$. Let $o \in \mathbb{N}$ be the offset of b_* and b'_* from the starting location of $src_1(b_*)$. We know $o + |\mathtt{t}| > s_{src}$, where s_{src} is the size of $src_1(b_*)$. By induction of lemma 8 on $G_1^\natural$, with $\mathtt{isValid}(G_1^\natural, b_*)$, we can infer $\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \mathsf{source}(\tau_{src})$. Together with $\mathsf{cast}(|\mathtt{t}|, \tau_*, \tau_2)$,

$$\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} \hookrightarrow_{i, i+|\mathtt{t}|} \tau_2 \tag{3.57}$$

Since $s_{src} \sqsubseteq \mathsf{size}(\tau_{src})$ and $o + |\mathtt{t}| > s_{src}$, according to (3.9), $\mathsf{size}(\tau_{src}) = \top$. From the resolution rule $\textsc{Scalar}$ and $\textsc{Collapse1}$, (3.57) can be rewritten as

$$\forall \tau_{src} \in \mathbb{C} \,.\, src_1(b_*) \in part_1(\tau_{src}) \implies \tau_{src} = \tau_2 \tag{3.58}$$

For any block $b \in \mathcal{B}_1$ such that either $\mathtt{overlap}(b'_*, b)$, we know $src_1(b)$ and $src_1(b_*)$ share the same base address. By induction of lemma 9 on $G_1^\natural$, we know

$$\forall \tau_{src}, \tau'_{src} \in \mathbb{C} \,.\, \begin{pmatrix} src_1(b) \in part_1(\tau'_{src}) \wedge \\ src_1(b_*) \in part_1(\tau_{src}) \end{pmatrix} \implies \tau_{src} = \tau'_{src} \tag{3.59}$$

By induction of lemma 7 on $G_1^\natural$, $\mathtt{contains}(src_1(b), b)$. Considering $\mathsf{size}(\tau_{src}) = \top$, by induction of the lemma,

$$\forall \tau, \tau'_{src} \in \mathbb{C} \;.\; b \in part_1(\tau) \wedge src_1(b) \in part_1(\tau'_{src}) \implies \tau'_{src} = \tau \tag{3.60}$$

From (3.58), (3.59) and (3.60), (Cast-Obj) holds.

In $\textsc{Arith-Op}_2$, $\mathcal{B}_3 = \mathcal{B}_2 \cup \{b', b'_*\}$. b' is disjoint with other blocks. If $b'_* \in \mathcal{B}_2$ or its type `t` is not scalar, the lemma holds. Otherwise, `t` is scalar and b'_* is a newly added scalar block. With `t` is scalar, we know $\mathsf{scalar}(\tau_*)$ and b_* is a scalar block. Considering $\forall \tau_* \in \mathbb{C} \;.\; b_* \in part_2(\tau_*) \implies b'_* \in part_3(\tau_*)$, we need to show

$$\forall \begin{pmatrix} b \in \mathcal{B}_2, \\ \tau'_*, \tau_* \in \mathbb{C} \end{pmatrix} . \begin{pmatrix} b \in part_2(\tau'_*) \wedge \\ b_* \in part_2(\tau_*) \end{pmatrix} \wedge \begin{pmatrix} \texttt{overlap}(b, b'_*) \wedge \\ isScalar(type_2(b)) \end{pmatrix} \implies \tau'_* = \tau_* \tag{Arith$_2$-Obj}$$

By induction on lemma 7 on $G_2^\natural$, $\texttt{contains}(src_2(b_*), b_*)$. According to the relation between `contains` and `icontains`, $\texttt{icontains}(src_2(b_*), i_1, j_1, b_*)$ where $i_1, j_1 \in \mathbb{N}$. By induction of the lemma 4 on $G_2^\natural$,

$$\forall \tau_{src}, \tau_* \in \mathbb{C} \;.\; \begin{pmatrix} b_* \in part_2(\tau_*) \wedge \\ src_2(b_*) \in part_2(\tau_{src}) \end{pmatrix} \implies \begin{pmatrix} \tau_{src} = \tau_* \vee \\ \tau_{src} \hookrightarrow_{i_1, j_1} \tau_* \end{pmatrix} \tag{3.61}$$

With lemma 3 and the constraint generation rule [$\textsc{Arith-Op}$], we have $\tau \rightharpoonup \tau_*$ and $\mathsf{collapsed}(\tau)$. With (3.13), (3.61) can be rewritten as

$$\forall \tau_{src}, \tau_* \in \mathbb{C} \;.\; src_2(b_*) \in part_2(\tau_{src}) \wedge b_* \in part_2(\tau_*) \implies \tau_{src} = \tau_* \tag{3.62}$$

By induction of lemma 7 on $G_2^\natural$, $\texttt{contains}(src_2(b), b)$. According to the relation between `contains` and `icontains`, $\texttt{icontains}(src_2(b), i_2, j_2, b)$ where $i_2, j_2 \in \mathbb{N}$, according to lemma 4,

$$\forall \tau'_{src}, \tau'_* \in \mathbb{C} \; . \begin{pmatrix} b \in part_2(\tau'_*) \wedge \\ src_2(b) \in part_2(\tau'_{src}) \end{pmatrix} \implies \begin{pmatrix} \tau'_{src} = \tau'_* \vee \\ \tau'_{src} \hookrightarrow_{i_2, j_2} \tau'_* \end{pmatrix} \tag{3.63}$$

From $\texttt{overlap}(b, b'_*)$, we know $src_2(b)$ and $src_2(b_*)$ share the base address. By induction of lemma 9 on $G_2^\natural$,

$$\forall \tau_{src}, \tau'_{src} \in \mathbb{C}. \begin{pmatrix} src_2(b) \in part_2(\tau'_{src}) \wedge \\ src_2(b_*) \in part_2(\tau_{src}) \end{pmatrix} \implies \tau'_{src} = \tau_{src} \tag{3.64}$$

From (3.62) and (3.65), (3.63) can be rewritten as

$$\forall \tau_*, \tau'_* \in \mathbb{C} \; . \begin{pmatrix} b \in part_2(\tau'_*) \wedge \\ b_* \in part_2(\tau_*) \end{pmatrix} \implies \tau_* = \tau'_* \vee \tau_* \hookrightarrow_{i_2, j_2} \tau'_* \tag{3.65}$$

Considering $\mathsf{scalar}(\tau_*)$, according to the resolution rule Scalar, (3.65) can be rewritten as following formula and (Arith$_2$-Obj) holds.

$$\forall \tau_*, \tau'_* \in \mathbb{C} \; . \; b \in part_2(\tau'_*) \wedge b_* \in part_2(\tau_*) \implies \tau_* = \tau'_*$$

$\square$

Chapter 4

Partitioned Memory Models

With the analysis framework of memory partitioning and the points-to analysis algorithms, in this chapter, we focus on building our partitioned memory models. Section 4.1 reviews the different memory models and illustrates the novelties of the partitioned model. Section 4.2 introduces a family of partitioned memory models and illustrates their difference via an example. Finally, experimental results are presented in section 4.3.

4.1 Overview of Memory Models

Consider the C code in Fig. 4.1. We will look at how to model the code using the flat memory model, the Burstall memory model, and the partitioned memory model.

Flat model. In the flat model, a single array of bytes is used to track all memory operations, and each program variable is modeled as the content of some address in memory. Suppose M is the memory array, a is the *location* in M which stores

```c
int a;

void foo() {
  int *b = &a;
  *b = 0xFFF;
  char *c = (char *) b;
  *c = 0x0;
  assert(a != 0xFFF);
}
```

Figure 4.1: Sample code with type-unsafe pointer cast

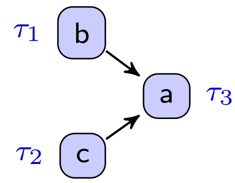

Figure 4.2: The points-to graph computed by Steensgaard's algorithm. Each τ_i represents a distinct alias group.

the value of the variable `a`, and b is the *location* in M which stores the value of the variable `b`. We can then model the first two lines of `foo` (following SMT-LIB syntax [3]) as follows:

```
(assert (= M1 (store M b a)))                   ; M[b] := a
(assert (= M2 (store M1 (select M1 b) #xfff)) ; M[M[b]] := 0xfff
```

This is typical of the flat model: each program statement layers another store on top of the current model of memory. When many statements are modeled, the depth of nested stores can get very large. Also, note that C guarantees that the addresses of `a` and `b` are not the same. The flat model must explicitly model this using an assumption on a and b. This can be done with the following disjointness predicate, where $size(p)$ is the size of the memory region starting at address p:

$$\mathsf{disjoint}(p, q) \equiv p + size(p) \leq q \vee q + size(q) \leq p.$$

For the code in Fig. 4.1, the required assumption is: $\mathsf{disjoint}(a, b) \wedge \mathsf{disjoint}(a, c) \wedge \mathsf{disjoint}(c, b)$. Deeply nested stores and the need for such disjointness assertions severely limit the scalability of the flat model.

Burstall model. In the Burstall model, memory is split into several arrays based on the *type* of the data being stored. In Fig. 4.1, there are four different types of data, so the model would use four arrays: M_{int}, M_{char}, M_{int*} and M_{char*}. In this model, a is a location in M_{int}, b is a location in M_{int*}, and c is a location in M_{char*}. Distinctness is guaranteed implicitly by the distinctness of the arrays. The depth of nested stores is also limited to the number of stores to locations having the same type rather than to the number of total stores to memory. Both of these features greatly improve performance. However, the model fails to prove the assertion in Fig. 4.1. The reason is that the assumption that pointers to different types never alias is incorrect for type-unsafe languages like C. In particular, b and c are aliases and should thus be located in the same array.

Partitioned model. In the partitioned memory model, memory is divided into regions based on alias information acquired by running a points-to analysis. The result of a points-to analysis is a points-to graph. The vertices of the graph are sets of program locations called alias groups. An edge from an alias group τ_1 to an alias group τ_2 indicates that dereferencing a location in τ_1 gives a location in τ_2. The points-to graph computed by Steensgaard's algorithm for the code in Fig. 4.1 is shown in Fig. 4.2. There are three alias groups identified: one for each of the variables `a`, `b`, and `c`. We can thus store the values for these variables in three different memory arrays (making their disjointness trivial). Note that according to the points-to graph, a dereference of either `b` or `c` must be modeled using the array containing the location of `a`, meaning that the model is sufficiently precise to prove the assertion.

```
typedef struct {F1 *next; uint32 idx;} F1;
typedef struct
 {F2 *next; uint16 idx1; uint16 idx2;} F2;

F1 f1; F2 f2;

void * bar(int flag) {
  F1 *p1 = &f1;
  p1->next = (F1 *)malloc(sizeof(F1));
  p1->idx = 0;
  p1->next->next = NULL;
  p1->next->idx = 1;

  F2 *p2 = &f2;
  p2->next = (F2 *)malloc(sizeof(F2));
  p2->idx1 = 0; p2->idx2 = 1;
  p2->next->next = NULL;
  p2->next->idx1 = 1;
  p2->next->idx2 = 0;

  void *p = (void *) (flag ? p1 : p2);
  return p;
}
```

Figure 4.3: Code with dynamic allocation and records

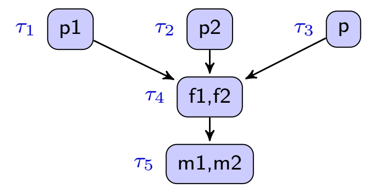

Figure 4.4: The field-insensitive points-to graph

4.2 Partitioned Memory Models

Consider now the code in Fig. 4.3, we use this example to introduce a family of partitioned memory models, presenting the points-to graphs and the details of modeling. This family includes the field-insensitive model, the field-sensitive model and the cell-based model.

Field-insensitive partitioned model. The field-insensitive partitioned model is based on Steensgaard's algorithm. We first note that dynamic memory allocation is modeled by introducing new variables `m1` and `m2` whose locations correspond to the results of calls to the first and second occurrences of `malloc`, respectively. Fig. 4.3 also includes record variables `f1` and `f2`. Steensgaard's original points-to analysis is field-insensitive, meaning that it always collapses all record fields into a single alias group, as shown in Fig. 4.4.

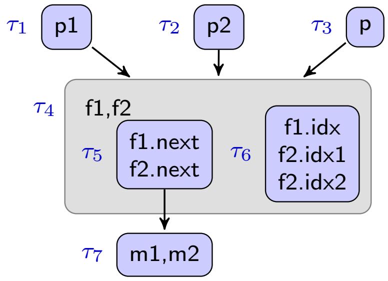

Figure 4.5: The field-sensitive points-to graph

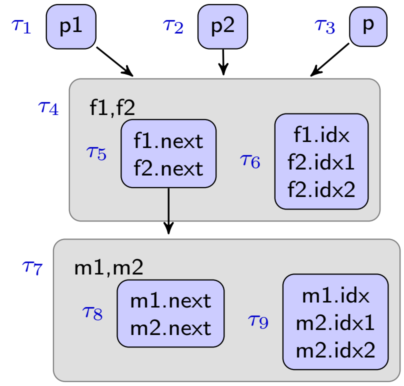

Figure 4.6: The cell-based field-sensitive points-to graph

Field-sensitive partitioned model. The field-sensitive partitioned model uses Steensgaard's field-sensitive points-to analysis, which builds the points-to graph shown in Fig. 4.5. Note that alias group τ_4, containing `f1` and `f2`, also contains two inner alias groups, denoted τ_5 and τ_6, representing the record fields of `f1` and `f2`.

Steensgaard's field-sensitive algorithm does more than just compute alias groups. It also computes the *size* of each alias group, which is either a numeric value (indicating the number of bytes occupied by every variable in that alias group) or $\top$, which indicates that the variables in the alias group have inconsistent or unknown sizes. This additional information enables further improvements in the memory model: the memory array for an alias group whose *size* is $\top$ is modeled as an array of bytes, while the memory array for a group whose *size* is some numeric value n can be modeled as an array of n-byte elements. For these latter arrays, it then becomes possible to read or write n bytes with a single array operation (whereas with an array of bytes, n operations are needed). Not having to decompose array accesses into byte-level operations reduces the size and complexity of the resulting

SMT formulas.

Cell-based partitioned model. Steensgaard's field-sensitive algorithm only distinguishes fields in statically allocated structured variables. Dynamically allocated structured regions are still collapsed into a single alias group. The new *cell-based* field-sensitive pointer analysis algorithm described in chapter 3 extends Steensgaard's algorithm to handle dynamically allocated regions and arrays more accurately. Fig. 4.6 shows the points-to graph computed by this analysis. Notice that τ_7 now resembles τ_4, with inner groups τ_8 and τ_9.

Another innovation is that our algorithm tracks the *cell size* (the size of each unit of access), rather than the *data size* of each alias group, making it possible to extend the approach mentioned above to handle both static and dynamic arrays. In particular, we can use a memory array whose elements are n bytes long to represent program arrays whose elements are of size n (bytes). This further improves the precision and performance of the memory model.

4.3 Evaluation

We implemented the memory models mentioned in this paper in the Cascade program verification framework [42]. A points-to analysis is run as a preprocessing step, and the resulting points-to graph is used during symbolic execution to: (i) determine the element size of the memory arrays; (ii) select which memory array to use for each read or write (as well as for each memory safety check); and (iii) add disjointness assumptions where needed (i.e. for distinct locations assigned to the same memory array).

To assess the effectiveness of the cell-based model, we conducted two experiments. In the first experiment, we compared the different memory models implemented in Cascade against each other. In the second experiment, we compared Cascade (using the cell-based model) with CBMC [24], LLBMC [17], Smack [21], SeaHorn [20], and CPAChecker [4]. We chose LLBMC, CBMC and Smack because, like Cascade, they rely on bounded model checking and satisfiability solvers. We chose SeaHorn because, like Cascade, it uses alias analysis (an invariant of DSA [29]) to infer disjoint heap regions. We chose CPAChecker because it was the winner of the memory safety category of the 2015 software verification competition (SVCOMP) [5]. All experiments were performed on an Intel Core i7 (3.7GHz) with 8GB RAM.

Benchmarks. In both experiments, we used a subset of the SVCOMP'16 benchmarks. We considered 141 benchmarks in the loops subset of the control flow category (Loops), the 81 benchmarks in the heap manipulation category (HeapReach), the and the 190 benchmarks in the heap memory safety category (HeapMemSafety), as these contained many programs with heap-allocated structures. For Loops and HeapReach, we checked for reachability of the `ERROR` label in the code. For HeapMemSafety, we checked for invalid memory dereferences, invalid memory frees, and memory leaks.

Configuration of Cascade. Like other bounded model checkers, Cascade relies on function inlining and loop unrolling. Cascade takes as parameters a function-inline depth d and a loop-unroll bound b. It then repeatedly runs its analysis, inlining all functions up to depth d, and using a set of successively larger unrolls until the bound b is reached. There are four possible results: *unknown* indicates that no result could be determined (for our experiments this happens only when the depth of function calls exceeds d); *unsafe* indicates that a violation was discovered; *safe* indicates that no violations exist *within the given loop unroll bound*; and *timeout* indicates the analysis could not be completed within the time limit provided. For the reachability benchmarks, we set $d = 6$ and $b = 1024$; for the memory safety benchmarks, we set $d = 8$ and $b = 200$. For these experiments, we used a strategy different from the one used for SVCOMP'16 (designed specifically for its scoring schema).

Comparison Table 4.1 reports results for the flat model, the partitioned model, the field-sensitive partitioned model (FS Partition), and the cell-based field-sensitive partitioned model (CFS Partition). In this table, "solved" means that either a violation was found or the maximum unroll was reached, and the time reported is the total for all solved problems. Fig. 4.7 shows scatterplots comparing each successive refinement. As can be seen, the partitioned model improves over the flat model. The field-sensitive partitioned model does help in some cases, but does worse on others. The cell-based model, in contrast, is nearly uniformly superior to both the partitioned model and the field-sensitive partitioned model.

	Loop(141)			HeapReach(81)			HeapMemSafety(190)		
	#solved	time(s)	ptsTo(s)	#solved	time(s)	ptsTo(s)	#solved	time(s)	ptsTo(s)
Flat	56	233	-	44	75.9	-	88	431.4	-
Partition	58	298.6	0.65	51	61.8	0.25	89	562.9	3.03
FS Partition	**60**	310.7	1.10	51	50.1	0.6	97	433.6	4.4
CFS Partition	59	226	0.87	**52**	47.5	0.82	**112**	627.2	9.96

Table 4.1: Comparison of various memory models in Cascade. The timeout is 60 seconds and the memory limit is 15GB. "ptsTo" is the time spent on the points-to analysis.

Table 4.2 compares the cell-based version of Cascade with LLBMC, CBMC, Smack, SeaHorn and CPAChecker. Note that Cascade is the only tool that does

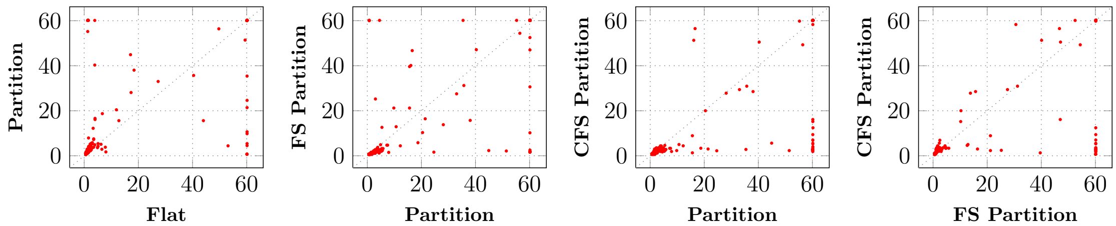

Figure 4.7: Comparison of various memory models on benchmarks of all three categories.

not produce false positives. For the HeapReach category, Cascade reports *unknown* for 20 benchmarks due to an insufficient function inline depth. Cascade performs best on the HeapMemSafety category and relatively good on the Loop category. While Cascade, solves more safe benchmarks than CPAChecker and SeaHorn in these categories, it should be noted that they are not bounded model checker. They use loop invariants to ensure soundness under certain assumptions.

	Loop				HeapReach				HeapMemSafety			
	Safe(93)		Unsafe(48)		Safe(56)		Unsafe(25)		Safe(105)		Unsafe(85)	
tool	#CA	#FP	#CA	#FN	#CA	#FP	#CA	#FN	#CA	#FP	#CA	#FN
Cascade	**93** 49407.8s	-	37 708.6s	11 5605.1s	42 7753.1s	-	19 60s	-	**101** 44467.5s	-	70 1596.2s	8 6805.7s
CBMC	**93** 14696.1s	-	38 362.9s	10 957s	**56** 17742s	-	**25** 136.7s	-	92 17918.2s	13 568.4s	65 169.7s	20 3880.2s
LLBMC	85 1220.3s	-	40 198.1s	4 1.91s	41 104.8s	-	22 10.68s	-	69 316.9s	2 9.5s	70 106.7s	7 187.1s
Smack	85 40159.7s	-	40 1150.3s	3 2651.3s	52 6383.7s	1 3.41s	**25** 146.7s	-	-	-	-	-
SeaHorn	77 1006.6s	8 13.6s	**43** 1570.3s	-	44 14.2s	11 4.8s	21 7s	3 1.7s	-	-	-	-
CPAChecker	56 3517.4s	1 456.9s	35 789.20s	-	45 1371.3s	1 4.2s	24 1264.3s	-	46 572.48s	3 10.5s	**72** 268.2s	-

Table 4.2: Comparison of Cascade (with CFS-Partition model), CBMC (CBMC-sv-comp-2016), LLBMC (llbmc-svcomp-14), Smack (smack-1.5.2-64), SeaHorn (SeaHorn v.0.1), CPAChecker (CPAchecker-1.4-svcomp16c-unix). "CA" is correct answer. "FP" is false positive (tool reports unsafe when it is safe). "FN" is false negative (tool reports safe when it is unsafe). The timeout is 900 seconds, and the memory limit is 15GB. For each category, the total number of benchmarks is shown in the followed parentheses. An entry of "-" means zero. The run time is shown in seconds under the number of benchmarks.

Chapter 5

Conclusion

This thesis is motivated by issues related to SMT-based program analysis and memory modeling in the low-level languages like C. We explore the idea of memory partitioning based on alias information, and propose a family of partitioned memory models. Our work is implemented in our static analysis tool Cascade, improving its scalability and its ability to discover critical memory safety bugs in benchmarks with complex data structures. In SV-COMP 2015, with the vanilla version of partitioned memory models, Cascade won the bronze medal in the memory safety category.

The development involves three components: the analysis framework supporting memory partitioning, the points-to analysis algorithms, and the modeling of memory. In chapter 2, we set up the analysis framework and formalized it with the approach of symbolic execution. The framework is built on the module of alias analysis module which can be instantiated with various analysis algorithms. In chapter 3, we presented a novel cell-based points-to analysis, which improves on the earlier field-sensitive points-to analysis by more precisely modeling arrays, unions, pointer casts, and dynamically allocated memory. The analysis was formalized in a constraint-based framework, and the proof of soundness was provided as part of the chapter. In chapter 4, we introduced a family of partitioned memory models built on various points-to analyses and showed how to use them to generate coarser and finer partitions. The experiments suggest that our cell-based memory model achieves both scalability and precision.

Overall, the partitioned memory model is a promising approach for modeling memory for program analysis in languages with the presence of pointers and low-level memory operations. Future work could involve introducing context-sensitivity and flow-sensitivity in order to obtain more precise partitioning.

Bibliography

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, May 1994.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. *CAV*, 6806:171–177, 2011.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard – version 2.0. In *Proceedings of the* 8^{th} *International Workshop on Satisfiability Modulo Theories (SMT '10)*, July 2010.

[4] D. Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. *CAV*, pages 184–190, 2011.

[5] Dirk Beyer. Software verification and verifiable witnesses (report on sv-comp 2015). *TACAS*, 2015.

[6] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. *Proceedings of Design Automation Conference (DAC'99)*, 317:226–320, 1999.

[7] S. Böhme and M. Moskal. Heaps and data structures: A challenge for automated provers. *N. Bjørner and V. Sofronie-Stokkermans, editors, Automated Deduction*, 6803:177–191, 2011.

[8] D. Brand and W. H. Joyner. Verification of protocols using symbolic execution. *Comput. Networks*, 2:351, 1978.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. OSDI 2008*, pages 209–224, 2008.

[10] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[11] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, 2009.

[12] E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for c. *ENTCS*, 254:85–103, 2009.

[13] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C a software analysis perspective. *SEFM*, 2012.

[14] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.

[15] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. *TACAS*, pages 337–340, 2008.

[16] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[17] S. Falke, F. Merz, and C. Sinz. LLBMC: Improved bounded model checking of c programs using llvm (competition contribution). *TACAS*, 7795:623–626, 2013.

[18] J. S. Foster, M. Fähndrich, and A. Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical report, U. OF CALIFORNIA, BERKELEY, 1997.

[19] R. Grimm. Rats!, a parser generator supporting extensible syntax. 2009.

[20] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *Proceedings of the 27th International Conference on Computer Aided Verification*, 2015.

[21] Arvind Haran, Montgomery Carter, Michael Emmi, Akash Lal, Shaz Qadeer, and Zvonimir Rakamaric. Smack+corral: A modular verifier (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, March 2015.

[22] M. Hind. Pointer analysis: Haven't we solved this problem yet? *PASTE 2001*, pages 54–61, 2001.

[23] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 385:226–394, 1976.

[24] D. Kroening and M. Tautschnig. CBMC - C bounded model checker - (competition contribution). *TACAS*, 8413:389–391, 2014.

[25] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[26] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *SIGPLAN Not.*, 47(6):193–204, June 2012.

[27] S. K. Lahiri and S. Qadeer. Back to the future. revisting precise program verification using SMT solvers. *POPL*, pages 171–182, 2008.

[28] A. Lal, S. Qadeer, and S. K. Lahiri. Corral: A solver for reachability modulo theories. *CAV*, 2012.

[29] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[30] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

[31] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[32] Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. *LCTES*, 2006.

[33] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer. ESBMC 1.22 (Competition Contribution). *TACAS*, 8413:405–407, 2014.

[34] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for c. *PASTE*, 2004.

[35] Z. Rakamarić and A. J. Hu. A scalable memory model for low-level code. *VMCAI*, pages 290–304, 2009.

[36] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[37] Nikhil Sethi and Clark Barrett. cascade: C assertion checker and deductive engine. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 166–169. Springer, 2006.

[38] Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 7–7, Berkeley, CA, USA, 2010. USENIX Association.

[39] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. *CC '96 Proceedings of the 6th International Conference on Compiler Construction*, pages 136–150, 1996.

[40] B. Steensgaard. Points-to analysis in almost linear time. *ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[41] M. Vujosevic-Janicic and V. Kuncak. Development and evaluation of LAV: an SMT-based error finding platform. *Proc. VSTTE*, 2012.

[42] W. Wang, C. Barrett, and T. Wies. Cascade 2.0. *VMCAI*, 2014.

[43] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. *PLDI*, pages 91–103, 1999.