



HAL
open science

A dependent nominal physical type system for the static analysis of memory in low level code

Julien Simonnet, Matthieu Lemerre, Mihaela Sighireanu

► To cite this version:

Julien Simonnet, Matthieu Lemerre, Mihaela Sighireanu. A dependent nominal physical type system for the static analysis of memory in low level code. 2023. hal-04649674

HAL Id: hal-04649674

<https://hal.science/hal-04649674v1>

Preprint submitted on 16 Jul 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Open licence - etalab

A Dependent Nominal Physical Type System for the Static Analysis of Memory in Low Level Code

JULIEN SIMONNET, CEA LIST, University Paris-Saclay, LMF, France

MATTHIEU LEMERRE, CEA LIST, France

MIHAELA SIGHIREANU, University Paris-Saclay, ENS Paris-Saclay and CNRS, LMF, France

We tackle the problem of *checking non-proof-carrying code*, i.e. automatically proving type-safety (implying in our type system spatial memory safety) of low-level C code or of machine code resulting from its compilation without modification. This requires a precise static analysis that we obtain by having a type system which (i) is expressive enough to encode common low-level idioms, like pointer arithmetic, discriminating variants by bit-stealing on aligned pointers, storing the size and the base address of a buffer in distinct parts of the memory, or records with flexible array members, among others; and (ii) can be embedded in an abstract interpreter. We propose a new type system that meets these criteria. The distinguishing feature of this type system is a nominal organization of contiguous memory regions, which (i) allows nesting, concatenation, union, and sharing parameters between regions; (ii) induces a lattice over sets of addresses from the type definitions; and (iii) permits updates to memory cells that change their type without requiring one to control aliasing. We provide a semantic model for our type system, which enables us to derive sound type checking rules by abstract interpretation, then to integrate these rules as an abstract domain in a standard flow-sensitive static analysis. Our experiments on various challenging benchmarks show that semantic type-checking using this expressive type system generally succeeds in proving type safety and spatial memory safety of C and machine code programs without modification, using only user-provided function prototypes.

1 INTRODUCTION

In heap-manipulating programs, one of the main issues is finding and preserving memory invariants describing the contents of heap cells, in particular when aliasing is allowed. The problem happens when a reference relies on an invariant on the contents of a memory region (i.e., a continuous sequence of bytes in memory), but this invariant is broken due to modification by another reference to or inside this region. For example, consider the encoding in C of an interval $[a, b]$ by a record `itv` with two integer fields. The internal invariant of `itv` is that $a \leq b$. A pointer `p` to a value of `itv` might expect this invariant to hold, but the invariant may be broken if a pointer `pb` to the second field at `p` is used to modify the value of `b` to a value strictly smaller than the first field `a`. A classical instance of this problem is the need to forbid polymorphic references in the ML language family [30, 76].

There are three main ways to deal with this problem. The first class of solutions, named *read-only* in the following, consists in just forbidding writing to memory cells (only the freshly allocated values can be stored). This is the main solution in pure functional programming languages, but read-only fields are also common in imperative languages. However, this solution is impractical for programs where in-place memory writes are important, e.g., for algorithmic reasons or for need of manual control over memory allocation.

The second class of solutions, named *aliasing control* in the following, tracks precisely all the references to a memory cell. For instance, separation logic [65] was designed to encode the “complex restrictions on the sharing of [data] structures”. Linear, uniqueness or ownership types [13] are used to limit the number of simultaneous references to an address. This class of solutions enables complex reasoning about the behavior of a program, including the ability to change the memory region invariant when it can be shown that no other references rely on the initial invariant (i.e., a

Authors' addresses: Julien Simonnet, CEA LIST, University Paris-Saclay, LMF, Palaiseau, France; Matthieu Lemerre, CEA LIST, Palaiseau, France; Mihaela Sighireanu, University Paris-Saclay, ENS Paris-Saclay and CNRS, LMF, 4 avenue des sciences, Gif-sur-Yvette, France.

50 *strong update* [73]). However, these solutions require complex reasoning on the program state. For
 51 example, they may require to track all the aliases to the changed memory region, which is difficult
 52 when the pattern used for sharing references is complex, like in garbage-collected programs or
 53 operating systems code. Another illustration of this difficulty is the need to abandon structural
 54 properties [75] when aliasing is controlled using a type system (e.g., using ownership types).

55 The third class of solutions does not restrict aliasing, but permits only heap modifications that
 56 *preserve the invariants* (i.e., *weak updates*). The drawback of this class is precision, because these
 57 solutions are unable to verify memory invariants that change over time, as needed for temporal
 58 memory safety. The benefit is simplicity, providing for simpler, tractable and efficient reasoning
 59 on data structures with complex sharing patterns, in particular by automated analyses that target
 60 spatial memory safety. Moreover, the memory is allocated in most programs on the heap to hold
 61 values of some type which is unchanged over time. This flow-insensitive heap-type relation is thus
 62 generally sufficient to prove spatial memory safety as a consequence of type safety in a suitable type
 63 system. Moreover, when use-after-free is impossible, e.g., when the memory is statically allocated
 64 or it is managed by a garbage collector based on tracing or reference counting, this invariant is
 65 enough to prove full memory safety.

66 In this article, **our main goal is to perform automated memory analyses using these type-**
 67 **based flow-insensitive memory invariants specified by the user, so as to prove type safety,**
 68 **implying spatial memory safety in our type system, in low-level heap-manipulating**
 69 **programs without modification of the source code.**

70 By low-level programs we mean programs implemented in low-level languages like C or machine
 71 code. The memory invariants in those programs must use low-level concepts, i.e., at the level of byte
 72 representation of values. Consequently, expressing them using types requires expressive features, in
 73 particular, *dependent types*. For instance, low-level arrays are encoded as a pointer (the array base)
 74 to a region whose size is given in another integer (the array size), which is a classic example of the
 75 use of dependent types. Furthermore, the bytes holding the array base may not adjoin the array size,
 76 illustrating the need for *non-local invariants* [33]. Another important feature is the frequent usage
 77 of *interior pointers* (pointing inside a record), and of functions operating over interior pointers of
 78 different datatypes as soon as part of the data layout is compatible (this is called *physical typing* in
 79 [10, 63]). Finally, low-level languages heavily make use of idioms allowing for efficient usage of
 80 the computer resources, such as *union types*, or taking advantage of pointer alignment to reuse the
 81 least significant bits in pointer values (*bit-stealing* [6]).

82 Our main contribution is **an expressive structural type system, designed to automatically**
 83 **prove spatial memory safety of low-level programs as type safety.** This type system uses
 84 a **nominal system of regions**, meaning that instead of having pointers to types, i.e., τ^* , which
 85 represents any memory address whose contents may be of type τ , our pointers are of the form
 86 η^* , where η is a *type name*. Then, pointers represent memory addresses which are labeled by a
 87 *region tag* obtained from η using an *allocation map* (e.g., bottom part of Fig. 1). In turn, the heap is
 88 constrained so that addresses tagged using η must contain values of the type τ defining η . Thus,
 89 region tags can be seen as a go-between a pointer and its pointed type.

90 Nominal systems of regions have already been used to verify imperative programs, e.g. [7, 14, 28,
 91 35, 60, 67]. However, these systems are generally *flat*, with no embedding or combination between
 92 memory regions. The main original feature of our type system is its ability to **combine regions**
 93 **by concatenation (record and array types), union (union and existential types), or sharing**
 94 **of parameters (global properties), in addition to refining regions (using refinement and**
 95 **intersection types).** This rich composition of memory regions is the key to handling the low-level
 96 programming patterns previously described, as we demonstrate in §2, without having to resort to ad-
 97 hoc types, tailored for the specific application, and typing rules as is often done [60, 69]. Furthermore,
 98

99 the combination between region composition and explicit naming brings up several interesting
100 properties. First, reasoning about points-to and aliasing properties (inclusion or disjunction of sets
101 of addresses) reduces to reasoning about the graph expressing how definitions of type names (η)
102 derive from each other (see §4). Second, a store can change the type of the value contained in a
103 memory region, provided a checkable condition that all region tags are compatible with this change
104 (see §6). We call this a *mild update* semantic of store because it combines the benefits of strong
105 update (the ability to change the type of a memory cell) and of weak update (there is no need to
106 track aliases) semantics.

107 This type system could be used as a typed assembly language [2, 21, 57, 78] and carried as a proof
108 [59] that the program is type-safe. The proof may be produced by a type-preserving compiler for a
109 type-safe language and checked using syntactic proof rules. However, our goal is to prove low-level
110 programs to be type-safe *automatically, without modification, even if the program was written in*
111 *an unsafe language like C or binary code*. Specifically, **we want to prove type-safe programs**
112 **written in type-unsafe languages like C, or programs transformed (using any tool) into**
113 **type-unsafe machine code, without any modification, given user-provided interface (type**
114 **definitions and function prototypes) using the types of our type system**. The absence of
115 code modification is important as modifying the code or executable may significantly worsen its
116 performance [70], and requires significant efforts which hinders adoption.

117 Relying on particular syntactic constructs to achieve this goal, as done in methods based on
118 syntactic type checking [14, 35, 60, 68], would defeat this goal of verifying the code unmodified,
119 and would probably make verification of machine code impossible. To achieve our goal and reach a
120 sufficient level of precision, **we make our type system and type checking algorithm semantic**
121 **using abstract interpretation** [17] (i.e., we formalize our type system semantically [2, 22, 51, 72]),
122 and implement it by a standard flow-sensitive static analysis. This semantic type-checking design
123 brings us at least three benefits. (i) It allows us to (try to) type check programs despite intrinsic
124 undecidability, with good results in practice. (ii) It provides us with a systematic method [18]
125 to state and prove the soundness of our typing rules, which corresponds to the static analysis
126 operations. (iii) It gives us modular extensibility, as the type checking can be seamlessly combined
127 with other numerical, memory, or control-flow analyses to improve its precision, overcoming the
128 syntactic limitations of purely syntactic type-checking methods. For instance, we can extend our
129 analysis with points-to predicates [61] that provides additional relations between an address and
130 its contents, and simultaneously makes use of the aliasing information provided by our types. We
131 can also use in parallel abstract domains for proving control safety by control-flow reconstruction
132 [5, 38], which allows us to remove from our type system the control-flow properties.

133 This focus on automation leads us to choose a structural type system which does not count the
134 number of references to a memory region. Indeed, the absence of reference counting reduces the
135 annotation burden by avoiding, e.g., Rust-like ownership and borrowing annotations. Furthermore,
136 the analysis will not fail because of imprecision in tracking the number of references. However,
137 **this choice prevents us to verify programs that perform arbitrary type-changing writes**.
138 This is a less strong limitation as one may think because (i) most writes are not type-changing
139 in C programs and (ii) some type-changing writes may be type-checked without sub-structural
140 rules using an expressive structural type system like TYPEDC (see example page 8). Still, **the main**
141 **limitation of our type-system is that it can not express nor prove temporal properties** (like
142 temporal memory safety or absence of data races) **or any property that requires a temporal**
143 **invariant on the heap** (like type-changing writes that are not in the class of mild updates), because
144 such properties require a substructural type system. However, our analysis is based on abstract
145 interpretation, a method which provides a framework for composition of analyses. Therefore,
146 by composing our analysis with those dealing with temporal properties (e.g., shape analysis or
147

Monat et al. method [54] for computing modular effects of threads), we obtain an alternative to sub-structural type systems. For instance, our analysis is combined with flow-sensitive abstractions of memory (e.g., the stack abstraction and the points-to predicates of Nicole et al. [61]) that allows us to infer some temporal memory invariants.

To demonstrate the applicability of our approach, we implement it in CODEX, an analyzer for C and binary code. We exercise CODEX on various low-level code of industrial quality (see §8), e.g., the binary code of a message passing library in an industrial micro-kernel [23], or parts of the runtime of the Lisp interpreter found in a compiled version of GNU Emacs [29], as well as on examples used by tools verifying spatial memory safety [66]. Our benchmark exhibits very complex sharing patterns, which would make approaches based on control of aliasing difficult to use; their low-level nature prevents their verification by a syntactic, flat nominal type system without rewriting and annotations. On the contrary, we show that our approach allows checking semantically type safety of such code with a very high degree of automation.

To sum up, our work makes the following contributions: (1) We formalize (§3) a rich dependent type system, including refinement types combined with records, pointers, arrays, type families, quantified and finite union types. (2) We tackle the classic unsoundness problem of strong updates over dependent types by proposing an original memory model (§5). (3) We propose a mild update semantic for store operation (§6). (4) We embed our type calculus into an abstract domain (§7) to obtain a sound semantic type checking. (5) We implemented our analysis for C and binary code and we demonstrate the effectiveness of our method (§8) on a challenging benchmark.

2 CHALLENGES FOR LOW-LEVEL SPATIAL MEMORY SAFETY

In a *flat* nominal systems of regions, featured in the Burstall-Bornat memory model [7, 8] as well as on many type systems for C [14, 28, 35, 60, 67], objects of different types are viewed as entirely separated (i.e., stored in disjoint regions of the heap). Such a system fails to verify spatial memory safety of low-level code, because it cannot accurately describe invariants on the shape of the memory used by such code. This section details three independent features that extend the flat nominal system of regions: relation between regions, union of regions, and concatenation/nesting of regions. We illustrate how our type system, by incorporating all these features, provides means of expressing memory invariants that accurately describe the common low-level code patterns, enabling their verification. Note that type definitions also translate to aliasing constraints, described in §5. *In the following examples, we suppose that integers and pointers have the same size, 4 bytes.*

Example 1: the need for dependence relations between regions. The C code on the right of Fig. 1 is extracted from the QDS [23] micro-kernel and simplified for readability; it encodes the message boxes used in the inter-process communication. The implicit invariant of the type `message_box` is that (a) the list starting at `first` is a circular non-empty list where (b) each element is a message with an allocated `buffer` of size `length` (stored once in `message_box`).

The property (b) requires the ability to encode **non-local invariants**, i.e., properties that relate memory regions that are not contiguous. For instance, the value of `length` field in `message_box` fixes the size of the region pointed by the `buffer` field in all `message` regions reachable from the `first` field. Non-local invariants are recognized as difficult to deal with in type-based memory analyses: e.g., [14, 33] are limited to local invariants, CHECKEDC requires the relation between an array length and the integer containing this length to follow specific syntactic patterns [28] to be able to add dynamic checks, etc.

Our type system allows us to express such invariants by means of parameterized type names, as shown in the specification given in the left part of Fig. 1. The specification includes the definition (introduced by `def` keyword) of a type name `message` parameterized by the integer `mLen`; the body

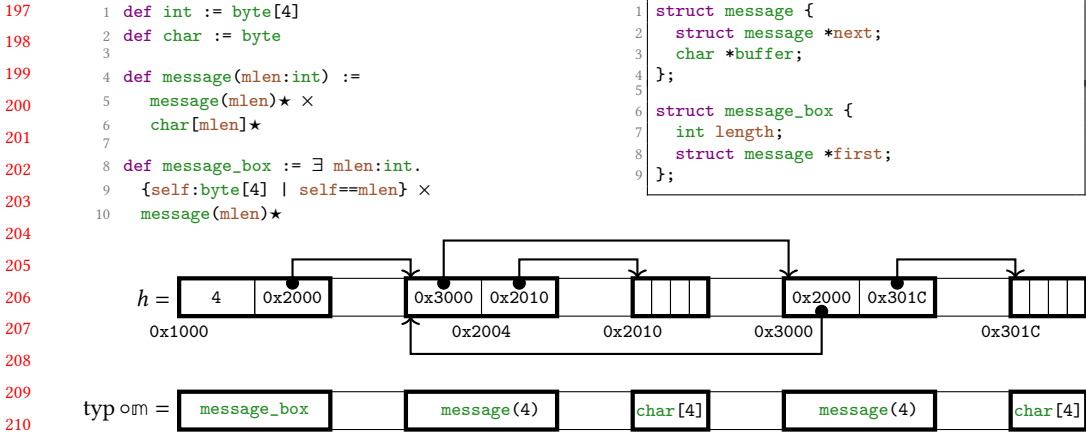


Fig. 1. Specification (left) of a data structure (right) in a micro-kernel and a possible memory layout (bottom)

of the definition of `message` is a record whose first field is a pointer to a `message` instantiated with the same `mlen` parameter and a pointer to a block of `mlen` characters. The `message` type name is used in the definition of the type name `message_box`, that employs existentially quantified types to introduce a variable linking the value of the first component of the product with the parameter of the second component. This is done using the refinement type “`{self:byte[4] | self==mlen}`” that denotes 4-byte values satisfying the given predicate. (Notice that, because our type system is nominal, `int` is not a synonym of `byte[4]`; nevertheless `==` is defined across both since they share the same set of values). Therefore, the specification of `message_box` denotes all the regions satisfying this internal invariant between their fields, which encodes the property (b).

The property (a) is obtained by associating to the $\eta\star$ notation the meaning of **not null pointer** since this property is important to prove spatial memory safety and mainly captured by the current memory analyses. The **possibly null pointers** are derived types in our type system, using a union with the null pointer type, as we will see in Fig. 3. Therefore, the specification in Fig. 1 (left) also expresses that the list of messages is not empty, the pointer to the buffer in each message is not null, and the list has a lasso shape¹, an over-approximation of the circularity.

To understand the meaning of these type specifications, we provide at the bottom of Fig. 1 an example for the memory layout when the values of these types are allocated in memory. The top array, labeled by h , represents the *heap*, mapping addresses to bit-vector values. The bottom array, labeled by typ om , is based on what we call an *allocation map* om , mapping the same addresses as h to the region tags specifying the values that the memory cells at these addresses should contain [74]. For the sake of space, we use type names instead of region tags (region tags are trees in our memory model, see §4.1), which is why we compose om with the typ function.

Example 2: the need for unions between regions. Fig. 2 (right) presents the C encoding of a core component of the Emacs-Lisp interpreter [29] simplified for the sake of readability. It defines “vector-like” objects (including real vectors, buffers, character sets, etc.) by employing a classical pattern to encode variant types in low-level programming: **discriminated unions**.² The union collects pointers to objects of different types, all of which share a common information (here the `header` field) that uses an integer value to discriminate between the different variants. For instance, a pointer of type `vectorlike_p` points to a `vector` (an array of Lisp values) if `header` is 0 or

¹Because of the finite number of memory addresses; however, an elaborate type definition may express the circular shape.

²Another way to encode variant types is using bit-stealing, as illustrated in the next example.

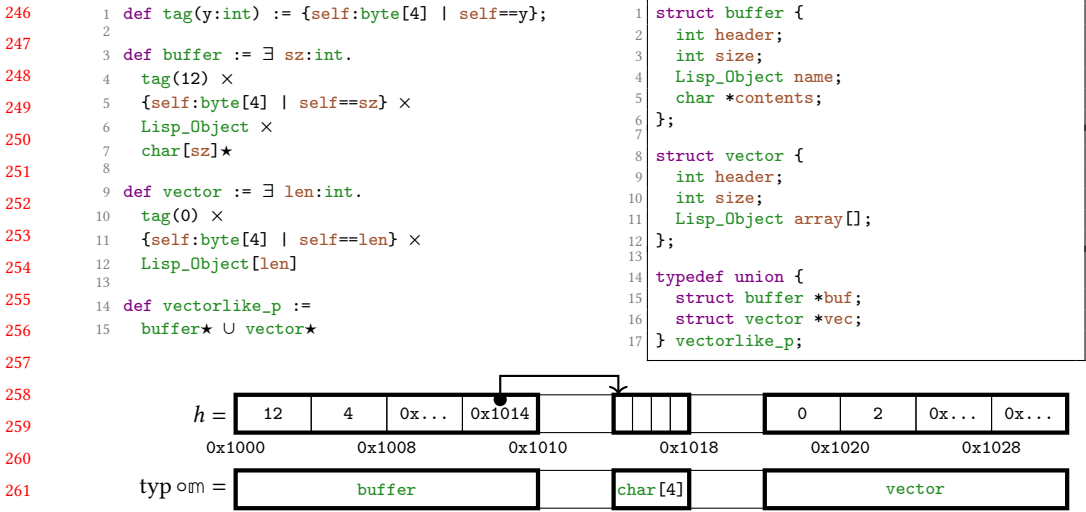


Fig. 2. Specification (left) for the data structure (right) used by the Emacs-Lisp interpreter for vector-like values and a possible memory layout (bottom)

to a `buffer` (an array of characters) if `header` is 12. Like in the previous example, the `buffer`'s content is stored in a different memory region whose size is stored inside the `size` field. However, the `vector`'s content is stored inside the `vector` value itself using the C feature known as **flexible array member**. In the specification at left of Fig. 2, this concurs with the fact that `char[sz]★` is a pointer type, unlike `Lisp_Object[len]`. *Union types* are used to represent non-discriminated unions, e.g., the values of type `vectorlike_p` may be either values of types `buffer★` or `vector★`. Finally, the *existential types* that we already saw actually represent an infinite union of regions, e.g., the `vector` type is the union of all the regions for every value of `len`.

Example 3: the need for concatenation and nesting of regions. The third example is extracted from the implementation of red black trees (RBT) in the Linux kernel [4] and it is presented in Fig. 3. It defines a C type `rb_node` as a **generic container** type carrying only RBT shape and color information. In C, this generic RBT is instantiated in `mytype` by *embedding* `rb_node`, along with the `key` (a pointer to a fixed-size string). We do the same in our specification, by defining `mytype` as a concatenation (using *product type*) of the `key` with `rb_node`. Note that the concatenation may also be described using *array types*, like in `char[16]`. The nesting of a region inside another implies that pointers to both regions may alias following relations described in §5.

The type `rb_node` uses **bit-stealing** to encode the color of a node in the last two bits of the pointer to the parent. We specify this property (left of Fig. 3) with the type name `myrb_node_color` using (i) a *refinement type* `rbcolor` encoding the type of two colors and (ii) an *existential type* `myrb_node_color` introducing the variables denoting the values of the pointer (α) and of the color (γ) combined to constrain the value of the first field of `myrb_node`.

In the specification, the generic C type `rb_node` is instantiated for the usage in `mytype` by defining `myrb_node`. This has two main consequences. First, the `myrb_node` type name precisely identifies the set of addresses storing `rb_node` values inside a `mytype` region. Second, the specification of pointers to `rb_node` inside `mytype` requires a special treatment. Describing the `rb_left` field of `myrb_node` as a possibly null `myrb_node` pointer is a sound description of the memory, but not precise enough: search algorithms depend on the fact that elements in the tree are **interior pointers** to `mytype` to recover the key from a pointer to a node, by subtracting a fixed offset (i.e.

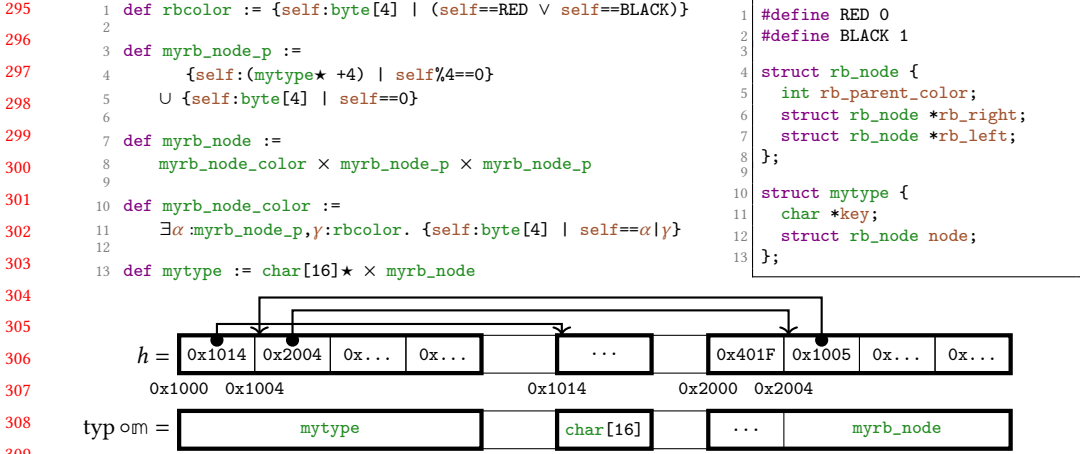


Fig. 3. Specification (left) of the C types for red black trees used in the Linux kernel and their usage (right) with a possible memory layout (at bottom)

4, the size of the key). We can specify this in the definition of `myrb_node_p` by adding 4 to a not null `mytype` pointer. Furthermore, we specify the **alignment of the pointer** necessary to do bit-stealing. Finally, these pointers may be null, and we express this using a *union* with a singleton type of 4 bytes for value 0. For instance, in the memory layout given at the bottom of Fig. 3, the value 0x1005 stored at address 0x2004 has type `myrb_node_color`, which describes a pointer to a `mytype` address 0x1000 shifted by 4 (0x1004), and the last bit is set because the node is BLACK.

Example 4: type changing writes with mild update. The C code at right of Fig. 4 defines a binary tree where each node maintains a reference to its parent (root's parent is 0). The three functions are memory safe because (i) they are called with non null references to nodes and (ii) the values of type `node` manipulated in the program are either a leaf node or of an interior node (`inode`) where both children are non null references (i.e., `node*`). These invariants are specified at left of Fig. 4; for sake of concision, we use the notations `inode?` and `nullptr` for the type expressions at right of \triangleq , i.e., possibly null pointer to `inode` and null pointer respectively.

A call to `extend` updates both children of the first argument `n` to non null references, thus changing the type of `n` from `node*` to `inode*`. Such a change of type is not allowed by a *weak update* semantics because this semantic requires that updates preserve the type *and* the case of values typed by union types. This constraint eases checking preservation of global memory's invariant since the type of every memory region is not changed. A *strong update* semantic allows to change the case of the union value in memory if it finds that the global memory is still well-typed (not necessarily by the types before the update). Such a check requires to keep track of pointers that may point inside the updated memory region. For instance, a pointer to `n->left` may change its type after `extend` from null to non null pointer.

One of our contributions is identifying a third way to allow safe updates changing the case of a union type without keeping track of pointer aliasing. We call this new semantic of updates *mild update*. Roughly, it allows non preserving writes of the union case if there are no pointers to the previous crashed type. This is the case in `expand` because `nullptr` value can not be aliased since `nullptr` is a type expression and not a type name, thus variable of type `nullptr*` can not be declared. The update done by a `to_leaf` call from an interior node to a leaf is also allowed by the mild update semantics for the same reason: the type `node*` can not be aliased. Notice that this


```

344 1  nullptr  $\triangleq$  {p:byte['W'] | p==0}
345 2  inode?  $\triangleq$  inode*  $\cup$  nullptr
346 3  def int := byte[4]
347 4  def inode := inode?
348 5       $\times$  int
349 6       $\times$  node*
350 7       $\times$  node*
351 8  def node := ( inode?
352 9       $\times$  int
353 10      $\times$  nullptr
354 11      $\times$  nullptr)
355 12      $\cup$  inode
356 13  inode* extend(node*, node*, node*);
357 14  node* to_leaf(node*);
358 15  node* next_right(node*);

```

```

1  typedef struct node_s node;
2  struct node_s {
3      node *parent;
4      int val;
5      node *right;
6      node *left;
7  };
8  node* extend(node *n, node *l, node *r) {
9      n->left = l; n->right = r;
10     l->parent = r->parent = n; return n;
11 }
12 node* to_leaf(node *n) {
13     n->left = n->right = 0; return n;
14 }
15 node* next_right(node *n) {
16     return (n->parent!=0)?n->parent->right:n;
17 }

```

Fig. 4. Specification (left) of the binary tree invariants used in the C code at right.

update of the union’s case is computed after both assignments of children to 0 are done. If only one child is changed to 0 at the end of the function, the analysis raises an alarm for bad type.

As a final remark, notice that our analysis proves that `next_right` is memory safe and returns a non null reference to `node`. This follows from `n` being a not null and therefore, its parent is an interior node with right child also not null.

3 PHYSICAL DEPENDENT TYPES

We start presenting the `TYPEDC` type system, by the definition of the set of type expressions on which it is built. The semantics of `TYPEDC` is presented in §4, the aliasing rules that it entails are in §5, and the core typing judgments are in §6. Throughout these sections we will use Fig. 5 as a running example; it contains a complete specification for two functions acting on a binary tree holding integer data. Each tree’s node is either an interior node with two children, or a leaf node with none. Each node except the root has a pointer to its parent, which is an interior node.

3.1 Syntax and intuitive semantics

Table 1 summarises the syntax of type expressions in `TYPEDC`. The set of free symbolic variables in a type expression or term is denoted by $fv(\cdot)$. A closed type/term has no free variables. A substitution of a free variable α by k in some type or term t is denoted by $t[\alpha \leftarrow k]$.

The type identifier `byte` is the predefined type for the smallest addressable memory piece in `TYPEDC`. Additional type names are introduced using a finite set of *type name definitions* Δ : each definition associates a type expression to a type identifier n and a (possibly empty) list of parameters. Type names η are either the predefined type identifier `byte` or the application of a type identifier n to a list of terms corresponding in size and type to the list of parameters in the definition of n . We denote by $\Delta(n(e_1, \dots, e_\ell))$ the type expression obtained by substituting the parameters in the definition of n by the values of terms e_1, \dots, e_ℓ respectively. We omit parentheses in type names when the list of parameters is empty. For instance, the definition of type name `tag` in Fig. 2 has one parameter, and it is instantiated in type `buffer` to obtain the type name `tag(12)`; an example of type name without parameters is `rbcolor` in Fig. 3. Additional constraints on the type definitions and their instantiation are exposed in §3.2. The set Δ is **an input of our analysis, provided by the user** after translation from a C-like surface syntax, which predefines standard C types like `int`, `long`, etc. Fig. 6 contains a complete example of a user-provided specification for Δ .

Pointer type expressions are built on type names. The simplest form is “ $\eta\star$ ”, which denotes the set of start addresses for memory regions whose tags include η in the memory layout (i.e., allocation

Table 1. Physical dependent types of `TYPEDC`

393	symbolic variables: $\alpha, \text{self} \in \mathcal{Q}$	integer constants: $\ell \in \mathbb{Z}$, bit-vector constants: $k \in \mathbb{V}$
394	type identifiers: $n \in \mathcal{N}$	operators: \diamond (bit-vector, comparison and logical)
395		
396		
397		
398	$\Delta \ni \text{def } n(\alpha_1 : \tau_1, \dots, \alpha_\ell : \tau_\ell) := \tau$	type name definition with $\text{fv}(\tau) \subseteq \{\alpha_1, \dots, \alpha_\ell\}$
399		
400	Type names $\eta ::= \text{byte} \mid n(e_1, \dots, e_\ell)$	
401	Types $\mathbb{T} \ni \tau ::= \eta \mid \eta \star + e \mid \{\text{self} : \tau \mid e\} \mid \tau_1 \times \tau_2 \mid \tau[e] \mid \tau_1 \cup \tau_2 \mid \exists \alpha : \tau_1. \tau_2 \mid \tau_1 \wedge \tau_2$	
402	Terms $\mathbb{E} \ni e ::= k \mid \alpha \mid e_1 \diamond e_2 \mid e_1 :: e_2 \mid e[k_1..k_2]$	
403		

map, see §4.1). To address a byte at an offset e inside a memory region labelled by η , we use type expression “ $\eta \star + e$ ”. The null address is not a value of $\eta \star$. However, possibly null pointers of type η , whose surface syntax is $\eta?$, are encoded using union and refined type expressions (see Fig. 3 and below). The values of pointer type occupy \mathcal{W} bytes, where \mathcal{W} is a constant fixed by the ABI. Like in §2, we suppose $\mathcal{W} = 4$ which also equals size of `int`. Notice that there are architecture-dependent aspects of the translation from the C-like surface language of specification (e.g., endianness) which come from the ABI parameter used by the analysis.

A *refinement type* expression “ $\{\text{self} : \tau \mid e\}$ ” specifies the set of values v of type τ such that the term e evaluates to true (i.e., $\neq 0$) when the symbolic variable `self` is replaced by v . For instance, `rbcolor` in Fig. 3 is a refined type. Notice that e need not to refer to the `self` variable, e.g., $\{\text{self} : \text{byte} \mid 1 == 0\}$ denotes a type of `byte`’s size with an empty set of values.

Product type expressions “ $\tau_1 \times \tau_2$ ” denote the bit-vector concatenation of a value in τ_1 with a value in τ_2 . They are used to specify record types in C, e.g., the `myrb_node` type in Fig. 3. We use the shorthand “ τ^n ” for a product of n types τ . The memory layout induced by the alignment constraints (i.e., padding) is made explicit using product types. For example a record type with two fields of type `byte` and `int` is specified by `byte × byte [3] × int` if `int` is 4-bytes.

Array type expressions “ $\tau[e]$ ” generalize the product to the concatenation of e values of type τ (e may be symbolic). Arrays with symbolic sizes are used to specify flexible array members like in the `vector` type on Fig. 2. To keep notations simple, we denote by `char[sz]★` (e.g., in type `buffer` from Fig. 2) a pointer to an array of `sz` characters, instead of `arr_char(sz)★` where `def arr_char(n:int) := char[sz]`.

Union type expressions “ $\tau_1 \cup \tau_2$ ” specify the union of sets of values of τ_1 and τ_2 . For example, the possibly null pointer notation $\eta?$ is a shorthand for $\eta \star \cup \{\text{self} : \text{byte}[\mathcal{W}] \mid \text{self} == 0\}$, which combines pointer, refinement, product and union type expressions. Notice that our union type is different from the sum type used in classic dependent types because it is not discriminated; e.g., union between types that share values is allowed.

Existential type expressions (or quantified union) “ $\exists \alpha : \tau_1. \tau_2$ ” generalize finite union types to an unbounded union³. Such types include values v of type $\tau_2[\alpha \leftarrow v_\alpha]$, i.e., τ_2 where v_α , a value of type τ_1 , replaces α . Without loss of generality, we suppose that all variables bound by existential quantifiers have unique names.

Intersection type expressions “ $\tau_1 \wedge \tau_2$ ” specify a restriction of values of τ_1 to the values of τ_2 , with τ_1 a union type including τ_2 as member; this expression is mainly used by the analysis.

Terms e are built on constants, symbolic variables in an unbounded set \mathcal{Q} , arithmetic operations, comparisons, logical operations (with non-null terms interpreted as true, and false otherwise), and bit-vector operations like concatenation “ $e_1 :: e_2$ ” and extraction “ $e[k_1..k_2]$ ” of bits of e ’s value

³Although similar to existential types in System F, our existential types quantify a value variable and not a type variable.

Table 2. Orderings \prec_{Δ} and \prec_{Δ}^{\exists} between type expressions.

$\Delta(\eta) \prec_{\Delta} \eta$	$\forall i \in \{1, 2\} : \tau_i \prec_{\Delta} \tau_1 \times \tau_2$	$\tau_1 \prec_{\Delta} \tau_1 \wedge \tau_2$
$\text{byte}[\mathcal{W}] \prec_{\Delta} \eta \star + e$	$\tau \prec_{\Delta} \tau[e]$	$\forall k \in \mathbb{V} : \tau_2[\alpha \leftarrow k] \prec_{\Delta} \exists \alpha : \tau_1. \tau_2$
$\tau \prec_{\Delta} \{x : \tau \mid e\}$	$\forall i \in \{1, 2\} : \tau_i \prec_{\Delta} \tau_1 \cup \tau_2$	$\tau_1 \prec_{\Delta}^{\exists} \exists \alpha : \tau_1. \tau_2$

between positions k_1 (included) and k_2 (excluded). Actually, terms may belong to any arithmetic theory with a sound decision procedure for entailment. In our analysis, we use the linear integer arithmetic with modulo constraints or bit-vector constraints, which is the logic theory underlying our numerical abstract domain (see §8).

3.2 Well-formed type definitions

Intuitively, $\text{def } \eta := \tau$ defines new kinds of contiguous memory regions “of type η ”. Specifically, a definition (i) constrains the possible values that can be stored in these regions (“regions of type η contain values of type τ ”), and (ii) it constrains the relations between the different regions (“regions of type η are a subset of the regions of type τ ”). The latter interpretation hints at a well-founded ordering between the tags of regions (provided in §5); this ordering derives from an ordering between type expressions, denoted \prec_{Δ} , which is formally defined in Table 2. The relation \prec_{Δ} is a strict preorder over type expressions that mostly corresponds to the sub-term relation. We explain in the following the differences with the sub-term relation. The rule $\Delta(\eta) \prec_{\Delta} \eta$ relates a type name with the expression given by its definition. Note that this rule implies that η and $\Delta(\eta)$ are not equivalent, which makes our type system *nominal*. The rule $\text{byte}[\mathcal{W}] \prec_{\Delta} \eta \star + e$ (and the absence of a rule $\eta \prec_{\Delta} \eta \star + e$) is another difference with the sub-term relation. Indeed, a pointer type $\eta \star + e$ does not “contain” values of type η , but it is a new scalar type holding an address. This rule is related to the C notion of forward reference to type names in pointer types. The absence of a rule $\tau_2 \prec_{\Delta} \tau_1 \wedge \tau_2$ also breaks the sub-term relation and comes from the asymmetric nature of our intersection. However, we require that τ_2 is related by the transitive closure of \prec_{Δ} to $\tau_1 \wedge \tau_2$:

REQUIREMENT 1 (WELL-FORMED INTERSECTION TYPES). *Any type expression $\tau_1 \wedge \tau_2$ satisfies $\tau_2 \prec_{\Delta}^* \tau_1$, i.e., type τ_2 is either τ_1 (degenerate case) or a type used in the definition of τ_1 .*

The rule $\forall k \in \mathbb{V} : \tau_2[\alpha \leftarrow k] \prec_{\Delta} \exists \alpha : \tau_1. \tau_2$ defines a sub-term ordering modulo the instantiation of the existentially quantified variable (whose possible values is over-approximated by \mathbb{V}). We don’t want a rule $\tau_1 \prec_{\Delta} \exists \alpha : \tau_1. \tau_2$ as the type $\exists \alpha : \tau_1. \tau_2$ may not contain values of the type τ_1 . However, we do want to prevent circular type definitions (except through a pointer type, like in C). So, we extend \prec_{Δ} with the relation \prec_{Δ}^{\exists} defined by $\tau_1 \prec_{\Delta}^{\exists} \exists \alpha : \tau_1. \tau_2$ and we ask the following:

REQUIREMENT 2 (WELL-FORMED Δ). *The relation $\prec_{\Delta} \cup \prec_{\Delta}^{\exists}$ induced by Δ is well-founded.*

The above requirement implies in particular the following property:

PROPOSITION 1. *The relation \prec_{Δ} is well-founded, i.e., if $\tau_1 \prec_{\Delta} \tau_2$ is understood as the successor relation $\tau_1 \rightarrow \tau_2$ in a graph, then \prec_{Δ} induces a DAG over type expressions.*

An example of such a graph is given in Fig. 5. The DAG induced by \prec_{Δ} (and its evolution in §5 that includes offsets) ensures type-based reasoning over aliasing [24] and separation. In particular, two pointers of type $\eta_1 \star + e_1$ and $\eta_2 \star + e_2$ such that neither $\eta_1 \prec_{\Delta}^* \eta_2$ nor $\eta_2 \prec_{\Delta}^* \eta_1$ do not alias. For instance, in Fig. 2, a pointer of type `buffer*` cannot be used to write into a region of type `vector`.

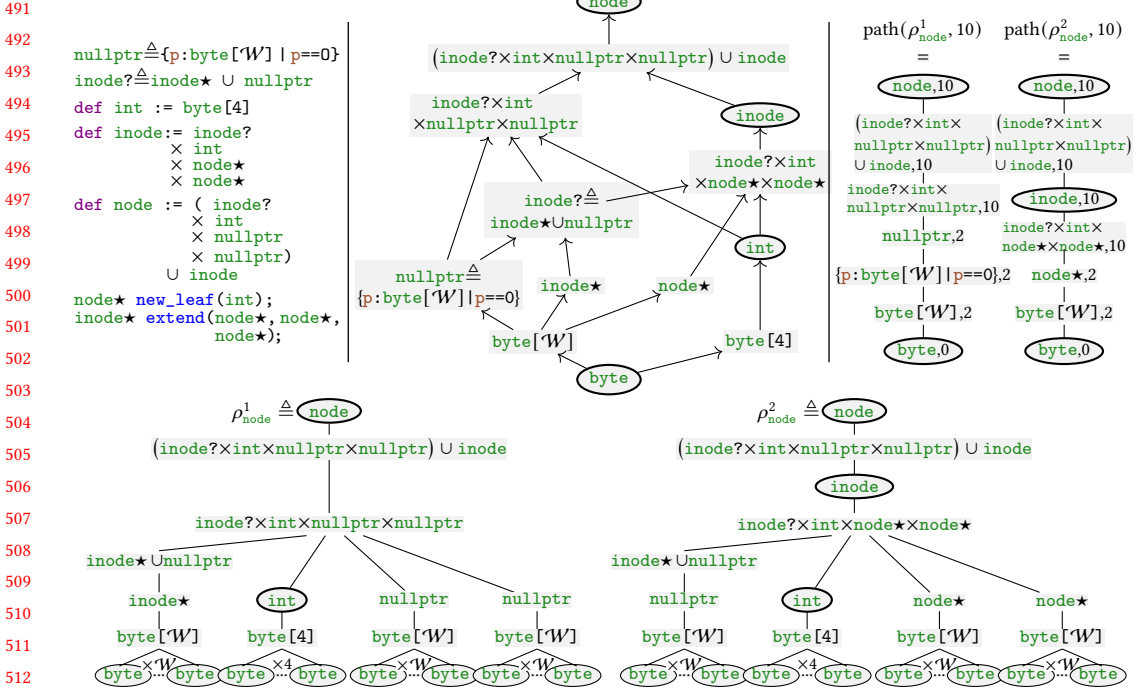


Fig. 5. Full specification (top left), DAG representation of \prec_{Δ} (top middle), two region tags for `node` out of 4 possible (bottom), and two paths (top right) in these region tags. Circled nodes are those whose type is a type name. The notations `nullptr` and `inode?` are *not* type names, but textually expanded notations used only for the sake of concision.

4 SEMANTICS OF PHYSICAL DEPENDENT TYPES

The interpretation of type expressions used by our type system depends on the memory model we consider. Usually, a memory model directly maps an allocated address to its type; thus, a pointer type τ^* represents all the start addresses of regions containing a value of type τ . Our type system is *nominal*: pointer types η^* can refer only to a type name η . Type names are used in our memory model to tag memory regions. This restriction on pointer types allows us in particular (i) to distinguish pointers to different regions even if these regions hold values of the same types (i.e., it enables a more precise alias analysis), and (ii) to change the value stored in a region without changing the type of pointers to this region, thereby allowing mild updates. More precisely, type names correspond to a set of region tags, where region tags identify memory regions and represent an invariant on the contents of a memory region (§4.2). Possible aliases between pointers are determined from the relations between region tags (§5). A region may switch between different region tags if some conditions are met (§6). We begin our exposition by presenting the memory model and the syntax of region tags.

4.1 Memory model

We first introduce some notations. We denote by $\langle \rangle$ the empty sequence, $\langle \sigma_1, \dots, \sigma_n \rangle$ is a sequence of n elements, $\vec{\sigma}$ is a sequence of $|\vec{\sigma}|$ elements with $\vec{\sigma}[i]$ the i th element of $\vec{\sigma}$. Concatenation $\vec{\sigma} \cdot \vec{\sigma}'$ and the last element $\text{last}(\vec{\sigma})$ are defined as usual. If σ_i appears in $\vec{\sigma}$, we denote it by $\sigma_i \in \vec{\sigma}$. The

notation $[\ell..u]$ denotes the set $\{i \mid \ell \leq i < u\} \subset \mathbb{Z}$ and \mathbb{V}_n is the set of n -bytes bit-vectors. $\text{dom}(f)$ and $\text{img}(f)$ respectively represents the domain and image of a function.

We consider a concrete memory model, formally defined in Table 3, where the set of addresses \mathbb{A} is a subset of \mathbb{V}_W . At each allocated address in the heap is stored a value in \mathbb{V}_1 , i.e., values of one byte. Because our analysis is based on a type abstraction of the memory, the concrete memory model also requires an *allocation map* \mathfrak{m} , which associates the first address in a range of n contiguous addresses, or *region*, to a type-based abstraction of the value stored. However, the abstraction used is not simply a type expression, but an ordered tree of type expressions which represents a possible memory layout for a type, called a *region tag* $\rho \in \mathbb{R}$. For instance the two region tags at the bottom of Fig. 5 represent two of the four possible layouts for `node` (the missing cases are when all the pointers are null, and when all are non-null).

In Table 3, region tags are defined as trees whose leaves are byte; the number of those leaves (i.e., width of the tree), is called region tag’s size. However, the structure of a region tag ρ in well-typed programs is constrained by the fact that ρ is one of the possible coherent memory layout for a type τ ; this constraint on *well-formed region tags* is formally stated as $\rho \in [\tau]_{\mathfrak{m}}$ with $[\cdot]_{\mathfrak{m}}$ defined in §4.2. Intuitively, a well-formed region tag is a tree whose root (computed by function `typ`) is a type name; all the children τ of a node τ' in the tree are such that $\tau' \prec_{\Delta} \tau$; each node has a single child, except record types $\tau_1 \times \dots \times \tau_n$ (that have n children), array types $\tau[e]$ (“value of e ” children), and byte (which are the leaves of the tree). In Fig. 5, both ρ_{node}^1 and ρ_{node}^2 are well-formed region tags.

Region tags relate bytes at different offsets: e.g., in Fig. 5, pointers to the children nodes are either both null or both non-null. We need to perform byte-level reasoning like this. To represent the k th byte in a memory layout ρ , we use a *byte tag* defined by the pair (ρ, k) . For instance, the start address of the second field of type `node` is denoted by $(\rho_{\text{node}}^1, 4)$ in Fig. 5 (for $W = 4$). From a given allocation map \mathfrak{m} , we define (in Table 3) the corresponding *byte allocation map* $\overline{\mathfrak{m}}$ as a mapping from an address a to its byte tag built from the tag $\mathfrak{m}(a_0)$ of the region starting at a_0 such that $a - a_0$ is positive and less than the size of the region (i.e., $\text{size}(\mathfrak{m}(a_0))$). The definition of allocation maps does not ensure the separation between regions. Indeed, two addresses a and a' such that $a < a'$ may be tagged by \mathfrak{m} such that the tag associated to a has a size (number of leaves) greater than $a' - a$, which means that the regions starting at a and a' are overlapping. Therefore, we require in the following sections⁴ that the allocation maps satisfy the following constraints ensuring region separation (in addition to well-formedness of region tags):

REQUIREMENT 3 (WELL-FORMED ALLOCATION MAP). *An allocation map \mathfrak{m} is well-formed iff $\forall \rho \in \text{img}(\mathfrak{m})$ the root of ρ , i.e., $\text{typ}(\rho)$, is a type name η ; ρ is well-formed, i.e., $\rho \in [\eta]_{\mathfrak{m}}$; and $\forall a_1, a_2 \in \text{dom}(\mathfrak{m})$ the regions $\mathfrak{m}(a_1)$ and $\mathfrak{m}(a_2)$ are separated, i.e., $a_1 + \text{size}(\mathfrak{m}(a_1)) \leq a_2 \vee a_2 + \text{size}(\mathfrak{m}(a_2)) \leq a_1$.*

4.2 Denotations of type expressions

For a given allocation map \mathfrak{m} , closed type expressions τ have a double interpretation in TYPEDC: as sets of values $\langle \tau \rangle_{\mathfrak{m}}$ and as sets of region tags $[\tau]_{\mathfrak{m}}$. These denotations are defined in Table 4 and the interpretation of a closed term e into a bit-vector denoted $\text{eval}(e)$ is done according to the semantics of C operators. Most of these definitions are obvious, we present the most interesting ones. Intuitively, the set of values of a pointer type $\langle \eta\star + e \rangle_{\mathfrak{m}}$ contains the addresses a where a belongs to a region starting at address a_0 , the region tag $\mathfrak{m}(a_0)$ contains a node η at some offset o , and the offset of a matches e (i.e. $a - a_0 = o + \text{eval}(e)$). For instance, in Fig. 5, if $\mathfrak{m}(0 \times 1000) = \rho_{\text{node}}^2$, then $0 \times 1006 \in \langle \text{int}\star + 2 \rangle_{\mathfrak{m}}$ because `int` appears at offset 4 in ρ_{node}^2 . Formally, an address a belongs to $\langle \eta\star + k \rangle_{\mathfrak{m}}$ if the pair (η, k) appears in $\text{path}(\overline{\mathfrak{m}}(a))$ defined as follows:

⁴In the following subsection of this section, allocation maps are arbitrary to avoid the definition of $[\cdot]_{\mathfrak{m}}$ to be circular.

Table 3. Memory model for TYPEDC.

589				
590				
591	Values	$v \in \mathbb{V}$	Addresses	$a \in \mathbb{A}, \mathbb{A} \subset (\mathbb{V} \cdot \mathcal{W} \setminus \{0\}) \subset \mathbb{V}$
592	Allocation maps	$\mathbb{M} \ni \mathbb{m} : \mathbb{A} \rightarrow \mathbb{R}$	Region tags	$\mathbb{R} \ni \rho ::= \text{byte} \mid \tau/\vec{\rho} \quad \text{for } \tau \neq \text{byte}, \tau \text{ closed}$
593	Byte allocation maps	$\overline{\mathbb{M}} \ni \overline{\mathbb{m}} : \mathbb{A} \rightarrow \overline{\mathbb{R}}$	Byte tags	$\overline{\mathbb{R}} \ni (\rho, k) \text{ s.t. } \rho \in \mathbb{R}, 0 \leq k < \text{size}(\rho)$
594	Type-offsets	$\mathbb{P} \triangleq \mathbb{T} \times \mathbb{N}$	Path in region tag	$\pi \in \mathbb{P}^+$
595				
596	$\text{size}(\rho) \triangleq$	$\begin{cases} 1 & \text{if } \rho \equiv \text{byte} \\ \overline{\text{size}(\vec{\rho})} & \text{if } \rho \equiv \tau/\vec{\rho} \end{cases}$	$\overline{\text{size}}(\langle \rho_0 \rangle \cdot \vec{\rho}) \triangleq$	$\text{size}(\rho_0) + \overline{\text{size}}(\vec{\rho})$
597			$\overline{\mathbb{m}}(a) \triangleq$	$(\mathbb{m}(a_0), a - a_0)$
598	$\text{typ}(\rho) \triangleq$	$\begin{cases} \text{byte} & \text{if } \rho \equiv \text{byte} \\ \tau & \text{if } \rho \equiv \tau/\vec{\rho}' \end{cases}$		if $a_0 \in \text{dom}(\mathbb{m})$ and
599				$0 \leq a - a_0 < \text{size}(\mathbb{m}(a_0))$

Table 4. Value denotation (left) and region tag denotation (right) of closed type expressions

600				
601				
602				
603	$(\cdot)_{\mathbb{m}} : \mathbb{T} \rightarrow \mathcal{P}(\mathbb{V})$		$[\cdot]_{\mathbb{m}} : \mathbb{T} \rightarrow \mathcal{P}(\mathbb{R})$	
604	$(\text{byte})_{\mathbb{m}} \triangleq \mathbb{V}_1$		$[\text{byte}]_{\mathbb{m}} \triangleq \{\text{byte}\}$	
605	$(\mathbb{n}(e_1, \dots))_{\mathbb{m}} \triangleq (\Delta(\mathbb{n}(e_1, \dots)))_{\mathbb{m}}$		$[\mathbb{n}(e_1, \dots)]_{\mathbb{m}} \triangleq \{\mathbb{n}(e_1, \dots)/\rho \mid \rho \in [\Delta(\mathbb{n}(e_1, \dots))]_{\mathbb{m}}\}$	
606	$(\eta\star + e)_{\mathbb{m}} \triangleq \{a \in \mathbb{A} \mid (\eta, \text{eval}(e)) \in \text{path}(\overline{\mathbb{m}}(a))\}$		$[\eta\star + e]_{\mathbb{m}} \triangleq \{(\eta\star + e)/\text{byte}[\mathcal{W}]\}$	
607	$(\{x : \tau \mid e\})_{\mathbb{m}} \triangleq \{v \in (\tau)_{\mathbb{m}} \mid \text{eval}(e[x \leftarrow v]) \neq 0\}$		$[\{x : \tau \mid e\}]_{\mathbb{m}} \triangleq \{(\{x : \tau \mid e\})/\rho \mid \rho \in [\tau]_{\mathbb{m}}\}$	
608	$(\tau_1 \times \tau_2)_{\mathbb{m}} \triangleq \{v_1 :: v_2 \mid v_i \in (\tau_i)_{\mathbb{m}}\}$		$[\tau_1 \times \tau_2]_{\mathbb{m}} \triangleq \{(\tau_1 \times \tau_2)/\langle \rho_1 \rangle \cdot \langle \rho_2 \rangle \mid \rho_i \in [\tau_i]_{\mathbb{m}}\}$	
609	$(\tau[e])_{\mathbb{m}} \triangleq \{v_0 :: \dots :: v_{s-1} \mid$		$[\tau[e]]_{\mathbb{m}} \triangleq \{(\tau[e])/\langle \rho_0 \rangle \cdot \dots \cdot \langle \rho_{s-1} \rangle \mid$	
610	$s = \text{eval}(e), \forall i \in [0..s], v_i \in (\tau)_{\mathbb{m}}\}$		$s = \text{eval}(e), \forall i \in [0..s]. \rho_i \in [\tau]_{\mathbb{m}}\}$	
611	$(\tau_1 \cup \tau_2)_{\mathbb{m}} \triangleq (\tau_1)_{\mathbb{m}} \cup (\tau_2)_{\mathbb{m}}$		$[\tau_1 \cup \tau_2]_{\mathbb{m}} \triangleq \{(\tau_1 \cup \tau_2)/\rho \mid \rho \in [\tau_1]_{\mathbb{m}} \cup [\tau_2]_{\mathbb{m}}\}$	
612	$(\exists \alpha : \tau_1. \tau_2)_{\mathbb{m}} \triangleq \bigcup_{v_1 \in (\tau_1)_{\mathbb{m}}} (\tau_2[\alpha \leftarrow v_1])_{\mathbb{m}}$		$[\exists \alpha : \tau_1. \tau_2]_{\mathbb{m}} \triangleq \{(\exists \alpha : \tau_1. \tau_2)/\rho \mid$	
613	$(\tau_1 \wedge \tau_2)_{\mathbb{m}} \triangleq (\tau_1)_{\mathbb{m}} \cap (\tau_2)_{\mathbb{m}}$		$\exists v \in (\tau_1)_{\mathbb{m}}. \rho \in [\tau_2[\alpha \leftarrow v]]_{\mathbb{m}}\}$	
614			$[\tau_1 \wedge \tau_2]_{\mathbb{m}} \triangleq \{(\tau_1 \wedge \tau_2)/\rho_1 \mid \rho_1 \in [\tau_1]_{\mathbb{m}} \wedge \tau_2 \in \rho_1\}$	
615				
616				
617				
618				

619 *Definition 4.1 (Paths in region tags).* Let ρ be a well-formed tag and $k \in [0.. \text{size}(\rho)]$ a constant.
620 The *path* inside ρ for offset k is computed by $\text{path} : \overline{\mathbb{R}} \rightarrow \mathbb{P}^+$ defined by:

$$621 \text{path}(\text{byte}, 0) \triangleq \langle (\text{byte}, 0) \rangle$$

$$622 \text{path}(\tau/(\vec{\rho}_1 \cdot \langle \rho \rangle \cdot \vec{\rho}_2), k) \triangleq \langle (\tau, k) \rangle \cdot \text{path}(\rho, k - \overline{\text{size}}(\vec{\rho}_1)) \quad \text{if } \overline{\text{size}}(\vec{\rho}_1) \leq k < \overline{\text{size}}(\vec{\rho}_1 \cdot \langle \rho \rangle)$$

623 The top-right part of Fig. 5 gives two paths for the byte tags $(\rho_{\text{node}}^1, 10)$ and $(\rho_{\text{node}}^2, 10)$ respectively.
624 The above definition and the one of $(\eta\star + k)_{\mathbb{m}}$ (Table 4) implies that this last set contains all addresses
625 tagged by $\overline{\mathbb{m}}$ with a (ρ, k') such that (η, k) is somewhere in the path of (ρ, k') (and not only at the
626 start of this path). If we continue our previous example where $\overline{\mathbb{m}}(0x1010) = (\rho_{\text{node}}^2, 10)$, we can see
627 that the address $0x1010$ belongs to all of $(\text{node}\star + 10)_{\mathbb{m}}$, $(\text{inode}\star + 10)_{\mathbb{m}}$, and $(\text{byte}\star + 0)_{\mathbb{m}}$.

628 The region tags of existential types are obtained by instantiating the quantified variable with
629 each value in its definition type. For an array type $\tau[e]$, a region tag is built from any combination
630 of region tags of its elements. The tags of intersection types $\tau_1 \wedge \tau_2$ recall the REQUIREMENT 1 for
631 well-formed intersection types stating that τ_2 is part of definition of τ_1 .

632 The values of a type $(\tau)_{\mathbb{m}}$ are not necessarily bit-vectors of the same size. For instance, consider
633 an array type where elements have a size fixed by an existentially quantified variable like in the
634 **vector** type in example from Fig. 2. However, C forbids type definitions where flexible size types

appear as a left member of a product and, by extension, as a type of elements in an array type. Therefore, the following hypothesis on type expressions will be frequent:

Definition 4.2 (Constant size type). A type τ has a constant size if there exists $s \in \mathbb{N}$ such that for any \mathfrak{m} well-formed, $\forall \rho \in \llbracket \tau \rrbracket_{\mathfrak{m}} : \text{size}(\rho) = s$. If τ has a constant size, we denote it by $\text{size}(\tau)$.

Definition 4.3 (Constant prefix hypothesis). The set of types \mathbb{T} satisfies the constant prefix hypothesis iff all its elements τ are such that:

- If τ is a product type $\tau_1 \times \tau_2$, then τ_1 has a constant size.
- If τ is an array type $\tau' [e]$, then τ' has a constant size.

This hypothesis allows simplifying some abstract operations in the analyzer, as we will now see.

5 THE DAG AND LATTICE OF TYPE-OFFSETS

The type definitions Δ and type expressions τ of `TYPEDC` are used to define two abstractions of addresses that help reasoning about aliasing: the DAG and lattice of type-offsets. Given an allocation map $\mathfrak{m} \in \mathbb{M}$, we saw that the denotation of pointer types $(\eta \star + e)_{\mathfrak{m}}$ provides a meaning to the pair of type name η and constant offset $\text{eval}(e)$. This can be generalized to any closed *type-offset pair* (TO) $(\tau, k) \in \mathbb{P} \triangleq \mathbb{T} \times \mathbb{N}$. We can view a type-offset pair (τ, k) as an abstraction of the set of addresses $\gamma_{\mathbb{P}, \mathfrak{m}}(\tau, k) \triangleq \{a \in \mathbb{A} \mid (\tau, k) \in \text{path}(\overline{\mathfrak{m}}(a))\}$. To obtain a sound abstraction, $\gamma_{\mathbb{P}, \mathfrak{m}}$ has to be monotonic [18]; this allows us to derive the *lattice of type-offsets*, detailed later in this section. Observe that a concrete address a goes through 3 successive levels of abstractions: (i) the byte tag $(\rho, k) = \overline{\mathfrak{m}}(a)$, (ii) the path $\pi = \text{path}(\rho, k)$, and finally (iii) one of the type-offset (τ, k') in π . In a type system with both unions and records, none of these abstractions are exact, thereby each is useful and needed. In particular, we also need to reason about paths (i.e., sequences of type-offsets), and for this we rely on the DAG of type-offsets.

The path abstraction using the DAG of type-offsets. For a given allocation map $\mathfrak{m} \in \mathbb{M}$, we define in Table 5 a strict order relation, the *type-offset derivation* relation $\prec_{\mathbb{P}, \mathfrak{m}} \subseteq \mathbb{P} \times \mathbb{P}$. The graph $(\mathbb{P}, \prec_{\mathbb{P}, \mathfrak{m}})$ is called a *DAG of type-offsets*. Such a DAG is a sound abstraction of all the possible sequences in $\text{path}(\rho, k)$ for all well-formed $\rho \in \mathbb{R}$ and all k s.t. $0 \leq k < \text{size}(\rho)$.

Intuitively, the relation $\prec_{\mathbb{P}, \mathfrak{m}}$ follows the derivation relation \prec_{Δ} on type expressions (see Fig. 2), adding the offset k . The relation between offsets in $\prec_{\mathbb{P}, \mathfrak{m}}$ depends on the size of type expressions involved in product or array types, which is not always a constant in the presence of both union and record types. For this reason, we have two sets of rules: `TOPROD1`, `TOPROD2` and `TOARR` apply to type-offsets whose first component fails to satisfy the constant prefix hypothesis in Def. 4.3, while `TOPROD1CST`, `TOPROD2CST` and `TOARRCST` simplify the previous rules respectively when the hypothesis is met. We prove in App. A that $\prec_{\mathbb{P}, \mathfrak{m}}$ is well-founded when \prec_{Δ} is well-founded. Using $\prec_{\mathbb{P}, \mathfrak{m}}$, we define in Table 6 the set of paths in the DAG of type-offsets, $\mathbb{P}_{\mathfrak{m}}^+$, where each path ends in $(\text{byte}, 0)$ (as paths of well-formed region tags end in byte). We also prove in App. A that $\mathbb{P}_{\mathfrak{m}}^+$ over-approximates the set $\{\overline{\mathfrak{m}}(a) \mid a \in \text{dom}(\mathfrak{m})\}$ of paths in \mathfrak{m} . Finally, we define $\mathbb{P}_{\mathfrak{m}}$, which restricts \mathbb{P} to the type-offsets found in the DAG of type-offsets for \mathfrak{m} .

The lattice of type-offsets. For a given \mathfrak{m} , the *join semi-lattice of type-offsets* $TO_{\mathfrak{m}} = \langle \mathbb{P}_{\mathfrak{m}}, \sqsubseteq_{\mathbb{P}, \mathfrak{m}}, \sqcup_{\mathbb{P}, \mathfrak{m}} \rangle$ is used to concretize and to define the join and inclusion operations of our representation of addresses. The order relation $\sqsubseteq_{\mathbb{P}, \mathfrak{m}}$ is defined as the domination relation in the type-offset DAG induced by $\prec_{\mathbb{P}, \mathfrak{m}}$, whose only start node is $(\text{byte}, 0)$. This implies that $\sqsubseteq_{\mathbb{P}, \mathfrak{m}}$ is a tree relation of root $(\text{byte}, 0)$. Therefore, $(\tau_1, k_1) \sqcup_{\mathbb{P}, \mathfrak{m}} (\tau_2, k_2)$ is defined to be the least-common ancestor on this tree relation.

Table 5. Derivation relation $\prec_{\mathbb{P}_m}$ between closed type-offsets for a fixed m (% is bit-vectors' modulo operation)

$\frac{\text{TODEF}}{\eta \neq \text{byte}}}{(\Delta(\eta), k) \prec_{\mathbb{P}_m} (\eta, k)}$	$\frac{\text{TOPTR}}{(\text{byte}^{\mathcal{W}}, k) \prec_{\mathbb{P}_m} (\eta \star + e, k)}$	$\frac{\text{TOREF}}{(\tau, k) \prec_{\mathbb{P}_m} (\{\text{self} : \tau \mid e\}, k)}$
$\frac{\text{TOUNION1}}{(\tau_1, k) \prec_{\mathbb{P}_m} (\tau_1 \cup \tau_2, k)}$	$\frac{\text{TOUNION2}}{(\tau_2, k) \prec_{\mathbb{P}_m} (\tau_1 \cup \tau_2, k)}$	$\frac{\text{TOINTER}}{(\tau_1, k) \prec_{\mathbb{P}_m} (\tau_1 \wedge \tau_2, k)}$
$\frac{\text{TOPROD1}}{\rho_1 \in [\tau_1]_m \quad \text{size}(\rho_1) > k}}{(\tau_1, k) \prec_{\mathbb{P}_m} (\tau_1 \times \tau_2, k)}$		$\frac{\text{TOPROD2}}{\rho_1 \in [\tau_1]_m \quad s = \text{size}(\rho_1) \leq k}}{(\tau_2, k - s) \prec_{\mathbb{P}_m} (\tau_1 \times \tau_2, k)}$
$\frac{\text{TOARR}}{\vec{\rho} \cdot \langle \rho' \rangle \in [\tau]_m^+ \quad \overline{\text{size}}(\vec{\rho}) \leq k < \overline{\text{size}}(\vec{\rho} \cdot \langle \rho' \rangle)}}{(\tau, k - \overline{\text{size}}(\vec{\rho})) \prec_{\mathbb{P}_m} (\tau[e], k)}$		$\frac{\text{TOEX}}{v \in (\tau_1)_m}}{(\tau_2[\alpha \leftarrow v], k) \prec_{\mathbb{P}_m} (\exists \alpha : \tau_1.\tau_2, k)}$
$\frac{\text{TOPROD1CST}}{\text{size}(\tau_1) > k}}{(\tau_1, k) \prec_{\mathbb{P}_m} (\tau_1 \times \tau_2, k)}$	$\frac{\text{TOPROD2CST}}{s = \text{size}(\tau_1) \leq k}}{(\tau_2, k - s) \prec_{\mathbb{P}_m} (\tau_1 \times \tau_2, k)}$	$\frac{\text{TOARRCST}}{s = \text{size}(\tau)}}{(\tau, k\%s) \prec_{\mathbb{P}_m} (\tau[e], k)}$

Table 6. Well-formed type-offsets and their concretization into sequences of type-offsets and addresses

$\mathbb{P}_m^+ \triangleq \{\pi \in \mathbb{P}^+ \mid \forall i. \pi[i+1] \prec_{\mathbb{P}_m} \pi[i] \wedge \text{last}(\pi) = (\text{byte}, 0)\}$	$\mathbb{P}_m \triangleq \{(\tau, k) \in \mathbb{P} \mid \exists \pi \in \mathbb{P}_m^+. (\tau, k) \in \pi\}$
$\gamma_{\mathbb{P}^+, m}(\tau, k) \triangleq \{\pi \in \mathbb{P}_m^+ \mid (\tau, k) \in \pi\}$	$\gamma_{\mathbb{P}, m}(\tau, k) \triangleq \{a \in \mathbb{A} \mid \overline{\text{m}}(a) \in \gamma_{\mathbb{P}^+, m}(\tau, k)\}$
	$\gamma_{\mathbb{P}}(\tau, k) \triangleq \{a \in \mathbb{A} \mid \exists m. a \in \gamma_{\mathbb{P}, m}(\tau, k)\}$

In Table 6, we decompose the definition of $\gamma_{\mathbb{P}, m}$ using the concretization $\gamma_{\mathbb{P}^+, m}$ of a type-offset into the set of paths that traverse it. This allows us to prove (see App. A) the following fundamental theorems for physical subtyping [10] and type-based alias analysis [24] in our type system.

THEOREM 5.1. *Let m be an allocation map and $(\tau_1, k_1), (\tau_2, k_2) \in \mathbb{P}_m$.*

- (1) *If $(\tau_1, k_1) \sqsubseteq_{\mathbb{P}_m} (\tau_2, k_2)$ then $\gamma_{\mathbb{P}, m}(\tau_1, k_1) \subseteq \gamma_{\mathbb{P}, m}(\tau_2, k_2)$, i.e., a dominating type-offset includes the addresses of the dominated type-offset.*
- (2) *If $\gamma_{\mathbb{P}^+, m}(\tau_1, k_1) \cap \gamma_{\mathbb{P}^+, m}(\tau_2, k_2) = \emptyset$ then $\gamma_{\mathbb{P}, m}(\tau_1, k_1) \cap \gamma_{\mathbb{P}, m}(\tau_2, k_2) = \emptyset$, i.e., if there is no path going through both (τ_1, k_1) and (τ_2, k_2) then they don't share any addresses.*

Example 5.2. For any m in Fig. 5, we have $(\text{int}, 0) \prec_{\mathbb{P}_m} (\text{inode}, 4)$ and $(\text{int}, 0) \prec_{\mathbb{P}_m} (\text{node}, 4)$. Because $(\text{int}, 0)$ is the least common predecessor of each in the type-offset DAG, it is the immediate dominator, and thus the result of $(\text{inode}, 4) \sqcup_{\mathbb{P}_m} (\text{node}, 4)$. Furthermore, pointers typed by $(\text{inode}, 4)$ and $(\text{node}, 8)$ do not alias as there is no path shared between these type-offsets.

Abstracting over allocation maps. Both relations defined above, $\prec_{\mathbb{P}_m}$ and $\sqsubseteq_{\mathbb{P}_m}$, depend on an allocation map. However, this allocation map is not known during the analysis. Therefore, we abstract it by defining $\prec_{\mathbb{P}} \triangleq \bigcup_{m \in \mathbb{M}} \prec_{\mathbb{P}_m}$. The direct definition of $\prec_{\mathbb{P}}$ is simple under the constant prefix hypothesis: just replace $\prec_{\mathbb{P}_m}$ by $\prec_{\mathbb{P}}$ in Table 5, and rewrite the premise of the rule TOEX as “ $\exists m \in \mathbb{M}$ well-formed, $v \in (\tau_1)_m$ ” to be independent of a fixed m . Reasoning about aliasing without the constant prefix hypothesis is possible but harder. Finally, the join semi-lattice $\langle \mathbb{P}, \sqsubseteq_{\mathbb{P}}, \sqcup_{\mathbb{P}} \rangle$ is defined by domination on the DAG induced by $\prec_{\mathbb{P}}$, like we did for $\langle \mathbb{P}_m, \sqsubseteq_{\mathbb{P}_m}, \sqcup_{\mathbb{P}_m} \rangle$.

6 CONCRETE SEMANTICS

The goal of our type system is to provide the following invariant on the heap: there exists an allocation map such that the heap is well-typed. This invariant, built on an enriched program semantics (detailed in App. B.2) that adds an allocation map to the heap, allows a static analysis to gain precision when values are loaded, but simultaneously requires the analysis to prove that stores maintain the invariant.

Concrete states. In an untyped execution semantic, program states s are pairs $(\sigma, h) \in \mathbb{S}$ of a store $\Sigma \ni \sigma : \mathbb{X} \rightarrow \mathbb{V}$ mapping program variables (including registers) in \mathbb{X} to values, and a heap $h : \mathbb{A} \rightarrow \mathbb{V}_1$ mapping allocated addresses to one byte values. We consider a typed semantics by constraining the values in the stack by a type and those in the heap by an allocation map \mathfrak{m} . Thus, for a given \mathfrak{m} , the set of values that can be stored at addresses tagged by $\rho \in \mathbb{R}$ is defined as follows:

$$\langle \text{byte} \rangle_{\mathfrak{m}} \triangleq \mathbb{V}_1 \quad \langle \tau / \vec{\rho} \rangle_{\mathfrak{m}} \triangleq \langle \tau \rangle_{\mathfrak{m}} \cap \{v_0 :: \dots :: v_{n-1} \mid v_i \in \langle \vec{\rho}[i] \rangle_{\mathfrak{m}}, i \in [0..n]\} \quad (1)$$

A *well-typed heap* is a pair (\mathfrak{m}, h) where \mathfrak{m} is well-formed, $\text{dom}(h) = \text{dom}(\overline{\mathfrak{m}})$ and for all $a \in \text{dom}(\mathfrak{m})$, $h(a) :: \dots :: h(a + \text{size}(\mathfrak{m}(a)) - 1) \in \langle \mathfrak{m}(a) \rangle_{\mathfrak{m}}$. Intuitively, well-typedness constrains the values in the heap to belong to those of the corresponding region in the allocation map.

Rules for load. Load rules use the above constraints to extract knowledge about the value being loaded. Let $a \in \mathbb{A}$ be an address and $\ell \in \mathbb{N}$ a strictly positive integer such that $[a..a + \ell] \in \text{dom}(h)$. We denote by $h[a..a + \ell]$ the *load from heap* operation which returns the bit vector of size ℓ obtained by concatenation of values $h(a) :: h(a + 1) :: \dots :: h(a + \ell - 1)$. The *store to heap* operation at address a on ℓ consecutive values with the new value $v \in \mathbb{V}_\ell$ is denoted by $h[a..a + \ell \leftarrow v]$.

PROPOSITION 2 (TYPED LOAD). *Let (\mathfrak{m}, h) be a well-typed heap and a be an address such that $[a..a + \ell] \subseteq \text{dom}(h)$ for some ℓ and $(\rho, k) = \overline{\mathfrak{m}}(a)$. Then $h[a..a + \ell] \in \{v[k..k + \ell] \mid v \in \langle \rho \rangle_{\mathfrak{m}}\}$.*

The above proposition enables inferring properties about the contents of the heap. Suppose a C function which receives a pointer \mathbf{p} of type `node*` (like `extend` in Fig. 5). If we load 4 bytes from $\mathbf{p}+8$, we know that we will receive a value v in $\langle \text{node*} \rangle_{\mathfrak{m}}$ or in $\langle \text{nullptr} \rangle_{\mathfrak{m}}$ (depending on the initial value of (\mathfrak{m}, h)). Furthermore, if the test v equals 0 succeeds, because $0 \notin \langle \text{node*} \rangle_{\mathfrak{m}}$ we deduce that the region corresponding to \mathbf{p} cannot have ρ_{node}^2 as a region tag, and that a value obtained by another load at $\mathbf{p}+12$ would also be in $\langle \text{nullptr} \rangle_{\mathfrak{m}}$ (i.e., would also be 0).

Rules for store and mild update. A store is safe if we can find a matching allocation map such that the new heap is well-typed, but also if existing typing judgments are preserved after the store.

Definition 6.1 (Safe store). Let (\mathfrak{m}, h) be a well-typed heap, $a \in \text{dom}(h)$ and $v \in \mathbb{V}_\ell$. The store $h[a..a + \ell \leftarrow v]$ is safe if there exists \mathfrak{m}' such that:

- (1) $(\mathfrak{m}', h[a..a + \ell \leftarrow v])$ is a well-typed heap, and
- (2) typing judgments are preserved: $\forall \tau \in \mathbb{T} : \forall v \in \mathbb{V} : v \in \langle \tau \rangle_{\mathfrak{m}} \Rightarrow v \in \langle \tau \rangle_{\mathfrak{m}'}$

Note that the program may perform a *partial store*, with $v \in \mathbb{V}_\ell$ at address a at some offset k inside a memory region of size s s.t. $s > k + \ell$. However, this partial store may require changing the region tag of the whole region. This case is dealt by the following theorem:

THEOREM 6.2 (SAFE STORE INSIDE A REGION). *Let (\mathfrak{m}, h) be a well-typed heap, $a \in \text{dom}(h)$ an allocated address such that $(\rho, k) \in \overline{\mathfrak{m}}(a)$ and $s = \text{size}(\rho)$. Let $v \in \mathbb{V}_\ell$ be a value. Then $h[a..a + \ell \leftarrow v]$ is a safe store iff $h[a - k..a - k + s \leftarrow h[a - k..a] :: v :: h[a + \ell..a - k + s]]$ is a safe store.*

Following the definitions in Table 4, the condition (2) of Def. 6.1 amounts to a monotony condition on the pointer types, i.e., $\forall \eta, k : \langle \eta \star + k \rangle_{\mathfrak{m}} \subseteq \langle \eta \star + k \rangle_{\mathfrak{m}'}$. This is easily proved when $\mathfrak{m}' = \mathfrak{m}$, which

Table 7. Definition of the abstract domains (top) and their meaning (bottom)

785	Table 7. Definition of the abstract domains (top) and their meaning (bottom)		
786	$\alpha \in \mathcal{Q}$ (symbolic variables)	$\hat{\mathbb{E}} \ni \hat{e} ::= \alpha \mid k \mid \hat{e} \diamond \hat{e} \mid \hat{e} :: \hat{e} \mid \hat{e}[k_1..k_2]$ (symbolic expressions)	
787	$v \in \mathcal{V} \triangleq \mathcal{Q} \rightarrow \mathbb{V}$ (valuation)	$\hat{\mathbb{T}} \ni \hat{\tau} ::= \eta \mid \hat{\tau} \star + \hat{e} \mid \{\text{self} : \hat{\tau} \mid \hat{e}\} \mid \hat{\tau}_1 \times \hat{\tau}_2 \mid \dots$ (symbolic types)	
788	$v^\# \in \mathcal{V}^\#$ (numerical domain)	$\Gamma^\# \in \Gamma^\# \triangleq \hat{\mathbb{E}} \rightarrow (\hat{\mathbb{T}} \times \hat{\mathbb{E}})$ (abstract type environment)	
789	$x \in \mathcal{X}$ (prog. variables)	$\sigma^\# \in \Sigma^\# \triangleq \mathcal{X} \rightarrow \hat{\mathbb{E}}$ (abstract store)	
790		$s^\# \in \mathcal{S}^\# ::= \Sigma^\# \times \Gamma^\# \times \mathcal{V}^\#$ (abstract state)	
791	$\gamma_{\hat{\mathbb{E}}} : \hat{\mathbb{E}} \rightarrow (\mathcal{V} \rightarrow \mathbb{V})$	$\gamma_{\hat{\mathbb{E}}}(\hat{e}) \triangleq \lambda v. \text{eval}(\text{subst}(\hat{e}, v))$	
792	$\gamma_{\hat{\mathbb{T}}} : \hat{\mathbb{T}} \rightarrow (\mathbb{M} \times \mathcal{V}) \rightarrow \mathcal{P}(\mathbb{V})$	$\gamma_{\hat{\mathbb{T}}}(\hat{\tau}) \triangleq \lambda(\mathfrak{m}, v). \llbracket \text{subst}(\hat{\tau}, v) \rrbracket_{\mathfrak{m}}$ with	
793		$\llbracket \tau \star + e \rrbracket_{\mathfrak{m}} \triangleq \{a \in \mathbb{A} \mid (\tau, \text{eval}(e)) \in \text{path}(\overline{\mathfrak{m}}(a))\}$	
794	$\gamma_{\Gamma^\#} : \Gamma^\# \rightarrow \mathbb{M} \rightarrow \mathcal{P}(\mathcal{V})$	$\gamma_{\Gamma^\#}(\Gamma^\#) \triangleq \lambda \mathfrak{m}. \bigcap_{(\hat{e}_1 \mapsto (\hat{\tau}, \hat{e}_2)) \in \Gamma^\#} \{v \in \mathcal{V} \mid \gamma_{\hat{\mathbb{E}}}(\hat{e}_1)(v) \in \gamma_{\hat{\mathbb{T}}}(\hat{\tau} \star + \hat{e}_2)(\mathfrak{m}, v)\}$	
795	$\gamma_{\mathcal{V}^\#} : \mathcal{V}^\# \rightarrow \mathcal{P}(\mathcal{V})$	(given by the numerical domain)	
796	$\gamma_{\Sigma^\#} : \Sigma^\# \rightarrow (\mathcal{V} \rightarrow \Sigma)$	$\gamma_{\Sigma^\#}(\sigma^\#) \triangleq \lambda v. \lambda x. \gamma_{\hat{\mathbb{E}}}(\sigma^\#(x))(v)$	
797	$\gamma_{\mathbb{M}^\#} : \Sigma^\# \rightarrow (\mathbb{M} \times \mathcal{V}) \rightarrow \mathcal{P}(\mathcal{S})$	$\gamma_{\mathbb{M}^\#}(\sigma^\#) \triangleq \lambda(\mathfrak{m}, v). \{(\sigma, h) \mid \sigma \in \gamma_{\Sigma^\#}(\sigma^\#)(v) \wedge (\mathfrak{m}, h) \text{ well-typed}\}$	
798	$\gamma_{\mathcal{S}^\#} : \mathcal{S}^\# \rightarrow \mathcal{P}(\mathcal{S})$	$\gamma_{\mathcal{S}^\#}(\sigma^\#, \Gamma^\#, v^\#) \triangleq \bigcup_{\mathfrak{m} \in \mathbb{M}, v \in \gamma_{\Gamma^\#}(\Gamma^\#)(\mathfrak{m}) \cap \gamma_{\mathcal{V}^\#}(v^\#)} \gamma_{\mathbb{M}^\#}(\sigma^\#)(\mathfrak{m}, v)$	
799			
800			
801			
802			
803			
804			
805			
806			
807	corresponds to a weak update. Arbitrary changes to \mathfrak{m} , as in a strong update (that we do not allow)		
808	would break this condition: given $a \in \llbracket \eta \star + k \rrbracket_{\mathfrak{m}}$, we can change $\mathfrak{m}(a)$ so that $a \notin \llbracket \eta \star + k \rrbracket_{\mathfrak{m}'}$ after		
809	the update. Consider a pointer p of type <code>inode\star</code> in Fig. 5. The judgment $p:\text{inode}\star$ would no longer		
810	be true if we wrote 0 at $p+8$ and $p+12$. Thus, because pointers to interior nodes exist, we cannot		
811	transform them into leaf nodes, as this would require a strong update.		
812	Mild updates are those that change \mathfrak{m} , the invariant on the contents of the heap, without breaking		
813	existing typing judgments. They are possible mainly because pointers are to type names, and not		
814	to arbitrary types. For instance, suppose that the function <code>extend</code> in Fig. 5 modifies a leaf node to		
815	turn it into an interior node by changing the two last fields from null to non-null pointers. As there		
816	is no name for the leaf node type, no “pointer to leaf node” type exists. Then, the fact that <code>extend</code>		
817	decreases the number of leaf nodes in \mathfrak{m}' is not a problem. We also have $\llbracket \text{node}\star \rrbracket_{\mathfrak{m}} = \llbracket \text{node}\star \rrbracket_{\mathfrak{m}'}$		
818	as the type <code>node\star</code> cannot distinguish a pointer to an interior node from a pointer to a leaf node.		
819	Finally, $\llbracket \text{inode}\star \rrbracket_{\mathfrak{m}'}$ increases, which is allowed. Thus, <code>extend</code> is well-typed.		
820	To verify the monotony, we just verify that the paths in the new allocation map \mathfrak{m}' contain all		
821	the type-offset pairs (η, k) , for which the type is a type name, that were in the previous allocation		
822	map \mathfrak{m} (i.e., all the paths must go through the same “circled nodes” in Fig. 5).		
823			
824	THEOREM 6.3 (NAME MATCHING IMPLIES MONOTONY). <i>Let $\mathfrak{m}, \mathfrak{m}' \in \mathbb{A} \rightarrow \mathbb{R}$.</i>		
825	<i>If</i>	$\forall a \in \text{dom}(\mathfrak{m}) : \{(\eta, k) \in \text{path}(\overline{\mathfrak{m}}(a))\} \subseteq \{(\eta, k) \in \text{path}(\overline{\mathfrak{m}'}(a))\},$	
826	<i>Then</i>	$\forall \tau \in \mathbb{T} : \llbracket \tau \rrbracket_{\mathfrak{m}} \subseteq \llbracket \tau \rrbracket_{\mathfrak{m}'}$ and $\llbracket \tau \rrbracket_{\mathfrak{m}} \subseteq \llbracket \tau \rrbracket_{\mathfrak{m}'}$.	
827			
828	For instance, suppose we try to transform an interior node at address p whose region tag is ρ_{node}^2		
829	to a leaf node. To make the resulting heap well-typed, the new region tag of p shall be ρ_{node}^1 . But		
830	$\text{path}(\rho_{\text{node}}^1, 10)$ in Fig. 5 does not contain the type-offset <code>(inode, 10)</code> , while $\text{path}(\rho_{\text{node}}^2, 10)$ does,		
831	so we reject this change. The opposite change, like in the <code>extend</code> function, makes every path go		
832	through more circled nodes, so it is well-typed.		
833			

7 TYPE-CHECKING BY ABSTRACT INTERPRETATION

Our analysis is defined as an abstract interpretation [17]. This section shortly presents the definition of our abstract domains and their meaning using concretization functions γ , some abstract transformers and their soundness. App. C details all abstract transformers and their properties.

7.1 Abstract domains

The main abstract domain of our analysis is the domain of abstract states \mathbb{S}^\sharp , which mimics the structure of concrete states \mathbb{S} (see §6). Table 7 summarizes the components of this main domain and their meaning using concretization functions γ . Below, we give an intuition on each component.

Symbolic expressions $\hat{\mathbb{E}}$ are expressions that may contain free variables in the set \mathcal{Q} of *symbolic variables*. A *valuation* $\nu \in \mathcal{V}$ is a mapping from symbolic variables to values. Given a valuation ν , we represent by $\text{subst}(\hat{e}, \nu)$ the constant term obtained from a symbolic expressions \hat{e} by substitution of the free variables in \hat{e} with their values in ν . This is used in the concretization $\gamma_{\hat{\mathbb{E}}}$: symbolic expressions are the abstract counterpart of constant values and closed terms in the concrete.

Symbolic types $\hat{\tau}$ extend the `TYPEDC`'s types $\tau \in \mathbb{T}$ defined in §3 to allow symbolic expressions instead of constant expressions, but also to permit extended pointer types of the form $\hat{\tau}\star + \hat{e}$ (where $\hat{\tau}$ is not needed to be a type name η) to improve precision of the analysis (see the discussion below). Therefore, we provide a definition for $(\tau\star + e)_{\mathfrak{m}}$ that extends Table 4. The concretization function $\gamma_{\hat{\mathbb{T}}}$ first concretizes a symbolic type $\hat{\tau}$ to a concrete type $\tau = \text{subst}(\hat{\tau}, \nu)$ given a valuation ν ; then, $\gamma_{\hat{\mathbb{T}}}$ uses the set of values $(\tau)_{\mathfrak{m}}$ for a given allocation map \mathfrak{m} . Thus, $\gamma_{\hat{\mathbb{T}}}$ is a function which inputs both a valuation ν and an allocation map \mathfrak{m} .

Abstract type environments Γ^\sharp have a role similar to type environments Γ in classical type theory, which map terms to types $\Gamma(e) = \tau$, and allow judgments $\Gamma \vdash e : \tau$. Similarly, our abstract type environments Γ^\sharp map symbolic expressions \hat{e} to a pair $(\hat{\tau}, \hat{e}')$ built from a symbolic type and a symbolic offset; furthermore, $\Gamma^\sharp(\hat{e}) = (\hat{\tau}, \hat{e}')$ implies the semantic judgment $s^\sharp \models \hat{e} : \hat{\tau}\star + \hat{e}'$ (where Γ^\sharp is a component of s^\sharp). Noteworthy, Γ^\sharp only relates expressions to pointer types because this is the only information which may not be represented by the numerical domain. Therefore, the static analysis gets us a flow-sensitive typing where the properties of the numerical expressions can be stored by the numerical domain ν^\sharp and retrieved using the typing rules when necessary. Because a symbolic expression \hat{e} is concretized to a value (by $\gamma_{\hat{\mathbb{E}}}$) using a valuation ν , the abstract type environment Γ^\sharp can be interpreted (using γ_{Γ^\sharp}) as a set of constraints on the valuations \mathcal{V} . For instance, if we derive using Γ^\sharp that $\alpha - 1$ is typed by `{self:byte[4] | self % 2 = 0}`, then the valuation ν is constrained such that $\nu(\alpha)$ is odd. The second argument of γ_{Γ^\sharp} is an allocation map \mathfrak{m} used to interpret symbolic types as sets of values.

As mentioned above, Γ^\sharp uses symbolic pointer types $\hat{\tau}\star + \hat{e}$ that extend the pointer types of `TYPEDC`. The reason is that the analysis may infer type judgments of the form $s^\sharp \models \hat{e} : \hat{\tau}\star + \hat{e}_2$ where $\hat{\tau}$ is not a type name (for instance in Fig. 5, if we have $s^\sharp \models \hat{e} : \text{node}\star$ and the value loaded at $\hat{e} + 8$ is null, we can infer that \hat{e} points to a leaf node, i.e. $s^\sharp \models \hat{e} : (\text{inode?} \times \text{int} \times \text{nullptr} \times \text{nullptr})\star$). One problem of the type judgments on extended types is that they may not be preserved by a mild update (see §6), unlike judgments of the form $s^\sharp \models \hat{e} : \eta\star + \hat{e}'$. Thus, in the formalization of abstract transformers (see App. C), we limit Γ^\sharp to contain mappings to pairs of the form (η, \hat{e}) . This means that judgments of the form $s^\sharp \models \hat{e} : \hat{\tau}\star + \hat{e}_2$ can be made but cannot be saved in Γ^\sharp , which makes sound the abstract semantics of store on heap operations.

The *numerical domain* \mathcal{V}^\sharp complements the set of constraints of Γ^\sharp . Elements of \mathcal{V}^\sharp are numerical constraints over symbolic terms (such as $3 \leq \alpha + 1 \leq 7$), and are interpreted by $\gamma_{\mathcal{V}^\sharp}$ as a set of

883 valuations that match all these constraints. All the classical numerical domains (e.g., intervals [17],
 884 congruences [31], octagons [53],...) can be used.

885 The memory is abstracted using *abstract stores* σ^\sharp , which map program variables \mathbb{X} to symbolic
 886 expressions $\hat{\mathbb{E}}$, an abstract counterpart to the concrete stores. Abstract stores are concretized by
 887 γ_{Σ^\sharp} to stores using a valuation of the symbolic expressions. There is no abstract counterpart to the
 888 concrete heap. Indeed, one of the goals of our type system is to define invariants on the memory that
 889 allow the program analysis without a representation of the heap (unlike, for instance shape analysis),
 890 in order to obtain fast analysis operations. However, given an abstract store σ^\sharp , an allocation map
 891 \mathfrak{m} and a valuation ν , we can define the set of all the possible corresponding concrete states (σ, h)
 892 using the function $\gamma_{\mathbb{M}^\sharp}$ (where \mathbb{M}^\sharp stands for memory). Noteworthy, the only constraint on the heap
 893 is that (\mathfrak{m}, h) is well-typed, which implies that $\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp)$ will be empty if \mathfrak{m} is not well-formed.

894 The *abstract state* combines store, typing and numerical abstractions $s^\sharp = (\sigma^\sharp, \Gamma^\sharp, \nu^\sharp)$. Their
 895 concretization $\gamma_{\mathbb{S}^\sharp}$ uses all the available elements in an abstract state to define the most precise set
 896 of possible states: given an allocation map \mathfrak{m} , we find valuations ν that fulfil the constraints given
 897 by Γ^\sharp and ν^\sharp , and we use them to build the set of all possible states using σ^\sharp .

898 This abstract state (and the analysis) may be easily extended with additional components. One
 899 particularly interesting addition are the points-to predicates [61], which partially represent the heap
 900 by relating pointer values to their contents. A reduced product between the points-to predicates
 901 and our domain allows using aliasing information obtained by types to know which points-to
 902 predicates are preserved by a store operation, or transferring typing judgments about the type of a
 903 pointed value to the type of the pointer.
 904

905 7.2 Flow-sensitive analysis

906 Our analysis is derived from the definition of our abstract domain and its concretisation, as usual in
 907 abstract interpretation [16, §45]. The basic transformers required by the abstract domain (reading
 908 from memory, storing in memory, computing expressions, testing conditions) involve additional
 909 operations to interchange information between the different parts of the analysis. We present all
 910 these operations and their soundness theorem in App. C. As an illustration, we provide in Table 8
 911 some of the rules necessary to handle the evaluation of expressions, including loading from memory,
 912 in a simple language whose semantics is close to that of machine code.

913 We adopt an axiomatic formalization of the abstract transformers, where different logical asser-
 914 tions correspond to different modules of the analyzer (in particular, abstract domains). For instance,
 915 the abstract evaluation of a program expression E into a symbolic expression \hat{e} [42] starting from
 916 the abstract state s_0^\sharp is denoted by the Hoare triple $\{s_0^\sharp\} E \Downarrow \hat{e} \{s_1^\sharp\}$. These assertions are defined
 917 by inference rules, so a proof tree may be used as a witness for type-safety [56, 59]. The actual
 918 implementation of the analysis (see §8) is a forward abstract interpretation derived from these
 919 rules: e.g., the implementation of $\{s_0^\sharp\} E \Downarrow \hat{e} \{s_1^\sharp\}$ is a function returning \hat{e} and s_1^\sharp from s_0^\sharp and E .
 920 Noteworthy, the implementation does not need to perform any backtracking proof search.

921 The evaluation of expression start by rules of the form $\{s_0^\sharp\} E \Downarrow \hat{e} \{s_1^\sharp\}$ done by a symbolic
 922 expression analysis domain [42] translating program expressions to symbolic expressions. When
 923 evaluating a division (rule BINOP /), the rule checks that the denominator is not null by querying
 924 the numerical abstract domain (assertion $s_2^\sharp \models \hat{e}_2 \neq 0$); an alarm is reported otherwise. The presence
 925 of such conditions make our type safety proof to imply the absence of runtime errors.
 926

927 When a new symbolic expression, like $\hat{e} + \hat{e}_2$, is created, new information is attached to it in
 928 the numerical or in the type domain. The rule PTRADD illustrates the case of the type domain,
 929 which infers symbolic types for symbolic expressions, i.e., assertions $s^\sharp \models \hat{e} : \hat{\tau}$. When a program
 930

Table 8. Some abstract transformers rules for expression evaluation and memory load

<p>CONST: $\frac{\{s_0^\#\} k \Downarrow k \{s_0^\#\}}{\{s_0^\#\} k \Downarrow k \{s_0^\#\}}$</p>	<p>VAR: $\frac{\{s_0^\#\} x \Downarrow s_0^\#. \sigma^\#[x] \{s_0^\#\}}{\{s_0^\#\} x \Downarrow s_0^\#. \sigma^\#[x] \{s_0^\#\}}$</p>	<p>LOAD: $\frac{\{s_0^\#\} E \Downarrow \hat{e}_1 \{s_1^\#\} \quad \{s_1^\#\} *_\ell \hat{e}_1 \Downarrow \hat{e}_2 \{s_2^\#\}}{\{s_0^\#\} *_\ell E \Downarrow \hat{e}_2 \{s_2^\#\}}$</p>
<p>BINOP \diamond: $\frac{\frac{\{s_0^\#\} E_1 \Downarrow \hat{e}_1 \{s_1^\#\} \quad \{s_1^\#\} E_2 \Downarrow \hat{e}_2 \{s_2^\#\} \quad s_2^\# \models \hat{e}_2 \neq 0 \text{ (if } \diamond = /)}{\{s_0^\#\} E_1 \diamond E_2 \Downarrow \hat{e}_1 \diamond \hat{e}_2 \{s_2^\#\}}}{\{s_0^\#\} E_1 \diamond E_2 \Downarrow \hat{e}_1 \diamond \hat{e}_2 \{s_2^\#\}}$</p>	<p>PTRADD: $\frac{s^\# \models \hat{e} : (\hat{\tau} \star + \hat{e}_1)}{s^\# \models (\hat{e} + \hat{e}_2) : \hat{\tau} \star + (\hat{e}_1 + \hat{e}_2)}$</p>	
<p>LOADSIMPLE: $\frac{s_0^\# \models \hat{e}_1 : (\hat{\tau} \star + \hat{e}_2) \quad s_0^\# \models \text{size}(\hat{\tau}) = \ell \quad s_0^\# \models \hat{e}_2 = 0 \quad \alpha \text{ fresh} \quad s_0^\# \wedge \alpha : \hat{\tau} \Rightarrow s_1^\#}{\{s_0^\#\} *_\ell \hat{e}_1 \Downarrow \alpha \{s_1^\#\}}$</p>		
<p>LOADLARGER: $\frac{s_0^\# \models \hat{e}_1 : (\hat{\tau} \star + \hat{e}_2) \quad s_0^\# \models \hat{e}_2 = k \quad s_0^\# \models \text{size}(\hat{\tau}) = \ell_2 \quad \ell_2 \geq k + \ell_1 \quad \{s_0^\#\} *_\ell_2 (\hat{e}_1 - k) \Downarrow \hat{e}_3 \{s_1^\#\}}{\{s_0^\#\} *_\ell_1 \hat{e}_1 \Downarrow \hat{e}_3 [k..k + \ell_1] \{s_1^\#\}}$</p>		

expression reads ℓ bytes at address E , denoted by $*_\ell E$ (rule LOAD), the evaluation involves memory domains. These domains handle assertions of the form $\{s_0^\#\} *_\ell \hat{e}_1 \Downarrow \hat{e}_2 \{s_1^\#\}$, meaning that the value read at the addresses given by \hat{e}_1 is the symbolic expression \hat{e}_2 . The rule LOADSIMPLE first queries the type domain to check if \hat{e}_1 is a pointer that allows loading ℓ bytes (this ensures memory safety of the load), then creates a fresh symbolic variable α and propagates the type information $\alpha : \tau$ to obtain new numerical information due to refinement types. Rule LOADLARGER applies when only a part of the target region is read. In this case, we load the full region and we extract the relevant part as the result.

The soundness of the analysis is done inductively by proving the soundness of each rule, i.e., we prove the soundness of the logical assertion in the conclusion from the soundness of the hypotheses. For instance, it is easy to prove sound the rule BINOP once we define the soundness of logical assertions appearing in this rule:

LEMMA 7.1 (SOUNDNESS OF EVAL). *Suppose $\{s_1^\#\} E \Downarrow \hat{e} \{s_2^\#\}$ and $s \in \gamma_{\mathbb{S}^\#}(s_1^\#)$ and $s \vdash E \Downarrow v$. Let (\mathfrak{m}, ν) such that $s \in \gamma_{\mathbb{M}^\#}(s_1^\#)(\mathfrak{m}, \nu)$. Then $s \in \gamma_{\mathbb{S}^\#}(s_2^\#)$ and $s \in \gamma_{\mathbb{M}^\#}(s_2^\#)(\mathfrak{m}, \nu)$ and $\gamma_{\mathbb{E}^\#}(\hat{e})(\nu) = v$.*

LEMMA 7.2 (NUMERICAL QUERIES). *If $s^\# \models \hat{e}$, $s \in \gamma_{\mathbb{S}^\#}(s^\#)$ and $s \in \gamma_{\mathbb{M}^\#}(s^\#)(\mathfrak{m}, \nu)$, then $\gamma_{\mathbb{E}^\#}(\hat{e})(\nu) \neq 0$.*

932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980

8 EVALUATION

This section presents the implementation of the TYPEDC type system in a static analyzer and its evaluation on a benchmark of binary and C programs. It aims to answer the following meta-question:

Is our approach based on semantic type-checking by abstract interpretation promising for proving spatial memory safety? We decompose this question in three research questions:

RQ1: Expressivity Is TYPEDC able to specify the low-level code patterns necessary to statically prove spatial memory safety of C and binary code?

RQ2: Performance Is the automatic analysis able to find bugs or to prove spatial memory safety with good performances in precision and time?

RQ3: Effort What is the specification effort, and how does it compare with the effort of state-of-the-art approaches to verify spatial memory safety like CHECKEDC [28, 49, 68]?

Implementation and experimental setup. We implemented our analysis as part of CODEX⁵, an open source analyzer for C and machine code written in OCaml. CODEX is a generic abstract interpreter able to detect different runtime errors (e.g. division by zero, illegal opcode, null pointer dereferences, etc.), here extended to also detect type-unsafe operations. It contains several abstract domains for value and memory analyses. We extended the inter-procedural analysis in CODEX with the abstract domain based on TYPEDC.

The analysis presented in §7 is faithful to our actual implementation, except for the following additional components. Firstly, the simple abstract stores $\sigma^\#$ are replaced with a domain more suitable to the languages we handle. For C, CODEX relies on the platform CPLATFORM to parse C files into a CFG and replaces $\sigma^\#$ with a flow-sensitive representation of addressable C local and global variables (similar to [52]). For machine code, CODEX relies on the platform BINSEC to translate machine code instructions into a simpler intermediate language, and replaces $\sigma^\#$ with a flow-sensitive abstraction of the stack, registers and global variables. CODEX also includes in $s^\#$ several abstract domains to reconstruct the control-flow graph [5, 38] during the analysis. In both C and machine code, our analysis employs points-to predicates similar to [61], but extended to support cross-refinement between points-to predicates and type judgments. An additional supporting abstract domain allows us to deal precisely with disjunctions of values coming from union types.

CODEX's inputs are (i) the (full C or binary) code, (ii) the entry point of the analysis given by the name of a function to be analyzed, and (iii) the type specification file (see below). The ABI parameter is fixed in our experimental setting to x86_32. CODEX outputs alarms when it detects: (a) an invalid memory access, e.g., out-of-bounds accesses to an array or structure or null pointer read or write; (b) a violation of a type specification, e.g., a load done at a non pointer value or a store with a value breaking the invariant of the specified type for the memory location; (c) a run-time error not concerning the memory, e.g., division by zero. Our test system uses an Intel Core i9-11950H machine with 32GB RAM running Ubuntu 22.04.

Benchmark selection. For our evaluation, we selected examples of C and binary code issued from real applications or existing challenging benchmarks used by static or dynamic methods for checking spatial memory safety. We split the examples in four parts, presented in Tables 9–10:

(1) OS includes QDS [23] (see Fig. 1), Contiki [27], and Linux RBTree [4] (see Fig. 3) which are excerpts from proprietary (QDS) or free OSes;

(2) Emacs includes functions from the Emacs Lisp run-time [29] version 27.2 (executable compiled by Debian); the functions manipulate various data structures (e.g., list, string, vector – see Fig. 2) in a particular or a generic way (e.g., length for number of elements in any kind of collection);

⁵The names of the analyzer and of supporting platforms have been anonymized. The tool is submitted for artifact evaluation.

(3) *Shapes* is used in [61] to challenge their analysis based on a simpler type system; it includes collections of various shapes benchmarks (e.g., linked lists, trees, graphs);

(4) *Olden* is a standard benchmark for tools checking spatial memory safety [14, 49, 58, 60, 66]. Each case study is superscripted by *bin* or *C* when only one kind of code is analyzed.⁶ In these examples, we observed the following challenging code patterns: (BS) bit-stealing, (DU) discriminated union for variant types, (NLI) non-local invariants, (FAM) flexible array member, (IP) interior pointers and (P?) possibly null pointer. Tables 9–10 indicate by ✓ the patterns found for each case.

Specification method. The results of the analysis depend on the specification of prototypes given as input. To write our specifications, we follow an iterative method that refines the C function prototypes produced by the standard *cproto* tool from the source code. We illustrate the C-like concrete syntax of the specification files on an example extracted from *Olden-bisort* case study (Table 10). The source code consists of 388 LoC and from it, *cproto* generates a specification file given in part on top of Fig. 6. With the generated specification, the analysis produces 9 (false) alarms. After reading the implementation of *RandTree*, we understood that the internal nodes always have two non-null children. We refine the specification of type *node* to introduce the union of two cases: both children null or both children non-null (we omit it for space reason). We also specify that the arguments of some functions cannot be a null pointer. In total, this adds 8 lines and modifies 6 lines of the original specification. By running our analysis with this new specification file,

```

1 typedef struct node {
2   int value;
3   struct node *left;
4   struct node *right;
5 } node;
6 ... // Other functions
7 node* RandTree(int, int, int, int);
8 int Bimerge(node*, int, int);
9 int Bisort(node*, int, int);

1 def node_pu(h) := union {
2   (node(h)+ with h>0) case1;
3   ((int with self==0) with h==0) case2;
4 };
5 def node(h) := struct {
6   int value;
7   node_pu(h-1) left;
8   node_pu(h-1) right;
9 };
10 def node_pp := ∃h:(int with self>0).node(h)+
11 def node_pz := ∃h:(int with self>0).node(h)?
12 ... // Other functions
13 node_pz RandTree(int, int, int, int);
14 int Bimerge(node_pp, int, int);
15 int Bisort(node_pp, int, int);

```

Fig. 6. Generated (top) and refined specifications.

we obtain one null-pointer dereference alarm in *Bimerge* function. This false alarm may be removed by adding a dynamic check in the code (which makes it type safe but may fail at runtime). It is better to remove it by further refining the specification. Indeed, the alarm points out to us that the *RandTree* function builds a fully balanced tree. To obtain this property, we change again the specification of *node* to introduce the height of nodes as a parameter because it is not stored in the nodes, as at the bottom of Fig. 6 (where we use + for ★). With this specification file as input, there are no remaining alarms, which implies that the code is spatially memory-safe. The effort of finding a specification (RQ3) is reduced by the help of the analyzer. It is reduced compared to manually introducing dynamic checks or developing special analyses in the compiler. This example and Tables 9–10 also demonstrate the expressivity of our type system (RQ1).

Comparison with existing approaches. The *Shape* benchmark from Table 9 allows us to compare indirectly with the shape analysis due to Nicole et al. [61]. Our times are comparable to their analysis times, which in turn are much better (RQ2) than the ones obtained by state-of-the-art tools used for shape analysis [11, 47, 48].

We also compare with CHECKED C [28, 49], a state-of-the-art tool to make C code spatially memory-safe. *Qualitatively*, one using CHECKED C has to perform three kinds of *code modifications*: (1) modification of the C code to handle limitations in the type system [49] like absence of unions,

⁶Some binary code is obtained by compiling existing C files, other – QDS, Emacs – is existing binary code.

Table 9. Experimental data for OS, Emacs and Shape benchmarks where **#LoC** is the number of commands; **#Entry** is the number of entry functions analyzed; **Spec** is the number of lines generated with `cproto` (gen) vs. manually changed (man) in type specification; **#Alarm** is the number of alarms with generated (gen) resp. manually changed (final) specification, and the (true) alarms identified with manual specification; **Time** is the global time (parsing the specification, parsing the code and running the analysis).

Case studies	#LoC	#Entry	Code patterns						Spec		#Alarms			Time (s)	
			BS	DU	NLI	FAM	IP	P?	gen	man	gen	final	true		
OS	Contiki	329	12	-	-	-	-	-	✓	19	14	16	2	0	1.33
	QDS ^{bin}	401	3	-	✓	✓	-	-	✓	83	83	18	0	0	1.28
	RBTree Linux	1 111	2	-	-	-	-	-	✓	29	17	6	2	0	0.46
Emacs	list ^{bin}	464	8	✓	✓	-	-	-	✓			-	0	0	3.03
	string ^{bin}	109	5	✓	✓	✓	-	-	✓		73	-	4	0	3.20
	buffer ^{bin}	42	3	✓	✓	-	✓	-	✓			-	0	0	3.12
Shapes	Graph	155	7	-	-	-	-	-	✓	26	14	0	0	0	0.79
	Javl	920	9	-	-	-	-	-	✓	37	34	10	1	1	0.70
	Kennedy	197	6	-	-	-	-	✓	✓	44	24	6	0	0	0.74
	RBtree	978	7	-	-	-	-	-	✓	32	18	56	16	0	0.42
	(6-)Other	5 742	19	-	-	-	-	-	✓	113	50	43	5	0	3.79

Table 10. Experimental data for Olden benchmarks used to compare with CHECKED C and 3C. Columns CC+3C: (man) are lines of code changed manually for CHECKED C, some of them (gen) may be inferred with 3C; *voronoi* is not dealt by CHECKED C. All alarms are eliminated by CHECKED C by using assertions.

Case studies (Olden)	#LoC	#Entry	Code patterns						CC+3C		Spec		#Alarms			Time (s)
			BS	DU	NLI	FAM	IP	P?	man	gen	man	gen	gen	final	true	
bh ^c	2 107	30	-	✓	-	-	-	✓	181	18	27	144	39	3	1	26.04
bisort ^c	356	11	-	✓	-	-	✓	✓	92	34	26	29	9	0	0	2.18
em3d ^c	693	17	-	-	✓	-	-	✓	158	88	52	53	42	15	0	6.48
health ^c	485	13	-	-	-	-	-	✓	99	57	39	57	16	4	0	5.96
mst ^c	431	5	-	✓	-	-	-	✓	161	28	17	44	33	10	3	1.89
perimeter ^c	486	12	-	✓	-	-	-	✓	44	10	69	41	13	1	0	1.64
power ^c	618	17	-	-	-	-	-	✓	83	20	26	75	26	5	0	6.04
treeadd ^c	249	2	-	-	-	-	-	✓	46	16	0	19	0	0	0	0.42
tsp ^c	617	12	-	-	✓	-	-	✓	78	10	2	32	6	0	0	3.86
voronoi ^c	1 151	40	✓	-	-	-	-	✓	✗	✗	38	101	57	44	0	21.35

or to comply with limitations of the type checker [28]; (2) insertion of annotations inside the code to help the type checker; (3) insertion of runtime checks (bound checks and null-pointer dereference checks) by the compiler, which may fail at runtime. Some of these code annotations may be generated automatically using 3C [49]. By contrast, we only require an external declaration of function prototypes, perform all checks statically, and do not change the implementation (the C and machine code is unmodified). *Quantitatively*, the columns **CC+3C** of Table 10 reproduce from [49] the number of changes made by CHECKED C and the ones that may be automatically generated by 3C. This shall be compared with the columns **Spec** which contain the number of lines of our specifications generated with `cproto` (gen) and manually added or changed (man). Columns **#Alarms** demonstrate that refining the specification avoids signaling false alarms (column final contains less alarms than column gen).

Conclusion. The added expressivity of our type system is necessary to represent the invariants of challenging code without modification (e.g., observe the ✓ in Table 10). The analysis is quite fast; there remains some imprecisions, but almost all of them are due to limitations of the analyzer implementation (discussed in §10), not to the expressivity of the type system. Furthermore, allowing the code to be modified would allow circumventing the current limitations of the analyzer. Finally, the semantic type checking requires a much lesser effort than syntactic methods like CHECKED C (observe, e.g., the `treeadd` benchmark). Most of the effort was spent reverse-engineering the type invariants, which would not be an issue if the tool were used during the development. Furthermore, our semantic analysis helped to discover real issues in the analyzed code (see App. D).

9 RELATED WORK

Memory models: A memory model is central to describe a programming language semantics so as to prove the soundness of analyses, be it type checking, abstract interpretation, or both as in our setting. High-level memory models, such as the Burstall-Bornat model [7, 8] where the memory for different types is entirely separate, or abstract models like the CompCert memory model [45, 46] that do not represent the value of pointers, cannot prove correct low-level operations such as casts between different types or modifying pointers using arithmetic or bitwise operations. Our memory model is concrete, i.e., corresponds to the machine semantics where all values, including addresses, are just bit-vectors. It is thus maximally expressive and can be used to prove sound low-level memory manipulations and the type and spatial memory safety of machine code. However, verifying programs in the more complex C semantics may require adding constraints like provenance [44, 50] to the model, to also be able to prove absence of miscompilation.

Semantic proofs of type soundness: The type-relevant aspects of our memory model simply consists in complementing the heap with an allocation map from addresses to tags, similarly to [74]. This simplicity is due to the fact that TYPEDC is nominal. Building semantic models of heaps with references to structural types (i.e., $\tau\star$ instead of $\eta\star$) is known to be a difficult issue [71]. Ahmed [1, §3.2.3] explains the core problem: if we model store typings as mappings from addresses to types, and types as predicates on store typings, we have a recursive definition which has an inconsistent cardinality. The complexity of the solutions to this recursivity problem (e.g., in [22, 72]) was one of the main reasons behind the rise of the syntactic method to prove type soundness [77]. For semantic proofs, a generic solution is step-indexed logical relations [1], which avoids the recursivity issue by stratifying the type system by indexing on the future evaluation steps. Such a solution would be difficult to employ for proving the soundness of operations in an abstract interpreter, where the canonical model (that we use) relates an abstract domain to a set of states. Thanks to our nominal type system, we propose a simple solution to this issue: pointer types point to region names instead of type expressions, so they do not depend on a store typing, but on an allocation map, that can be defined without types. Moreover, our semantic model is suited to use in an abstract interpreter.

Syntactic type-checking of low-level code: Several analyses have been proposed to prove memory safety of low-level programs using type-checking. Some of these analyses use type systems focused on the control of aliasing using ownership types or separation logic (e.g. [9, 12, 34, 69]), a style of reasoning complementary to ours that has advantages, but also makes it difficult to reason automatically on programs with arbitrary sharing of references.

Among type systems that have been proposed to prove spatial memory safety using a reasoning based on invariant preservation, typed assembly languages [2, 21, 57, 78] have been used to prove type safety of assembly code. However, they are tailored to prove the safety of a given type-safe source language where the type derivation is provided by the compiler, not for the case where the program is written in machine code and the type checking must be done automatically. Furthermore, in these systems, the pointers are structural, not nominal.

The line of work which is closest to ours employs type systems for verifying spatial memory safety of C-like languages. The main design of these systems relies on a combination between syntactic type checking and program transformation by insertion of runtime checks, which adds considerable overhead [58]. Furthermore, the runtime information needed by early systems like Cyclone [35] and CCured [60] changes the layout of types in memory, causing incompatibility. Deputy [14, 79] and CHECKEDC [28, 68] manage to maintain a compatible memory layout by using dependent types, which requires the user to annotate the code; this offloads runtime work to the static analysis phase. Our work continue in this area, by aiming to verify spatial memory safety fully statically, which means only specification overhead and full compatibility, as we do not modify

1177 the program. Some of these type systems support nominal nesting of structures, and Deputy also
1178 supports some variant types. It is also common to reason about structure nesting to derive aliasing
1179 information [24]. What we add in this area is our notions of region tags, byte tags, and the lattice
1180 of type offsets, that extends type-based reasoning about aliases to type systems allowing both
1181 concatenation and union of regions, and allows byte-level reasoning [52] about memory.

1182 **Run-time and hybrid type checking:** Some approaches [25, 37] propose run-time type checking
1183 for low-level languages like C/C++ to obtain memory safety and execution time. These approaches
1184 can be seen as coding a concrete semantics of our type system. Although they deal with temporal
1185 memory safety, they lack the expressivity of our type systems, especially non local invariants
1186 (specified by parameterized, existential and refined types), and they introduce execution overhead.

1187 **Type checking using abstract interpretation:** Flow-insensitive syntactic type checking suffers
1188 from inherent limitations [28] which are solved by annotating, porting the code, and inserting
1189 dynamic checks. To alleviate this problem, automated porting tools have been developed for
1190 CCured [60] and CHECKEDC [49, 68]. An alternative is to make the type checking algorithm more
1191 precise, which is necessary when we cannot modify the source code, like in machine code analysis.

1192 Harren [33] uses abstract interpretation to perform a flow-sensitive type checking for some
1193 dependent type system, which is necessary to verify assembly code. His tool cannot handle in
1194 particular non-local invariants and relies on the addition of ad-hoc type constructors (and typing
1195 rules) to handle low-level constructs like union types, which we address in our type system. Low-
1196 level liquid types [67] perform a flow-sensitive type inference, but the algorithm is not based on a
1197 standard abstract interpretation, which implies that it must make conservative decisions for some
1198 analysis steps (e.g., fold and unfolding of variables) that impact the precision of the analysis. Their
1199 memory structure is also limited to a flat system or regions.

1200 The analysis of Nicole et al. [61] is the closest to ours, as it also combines abstract interpretation,
1201 physical and refinement types to prove spatial memory safety. We extend their work by also
1202 supporting unions (using union and existential types) and relations (using parameterized types)
1203 between regions. This makes the type system, semantic model, and aliasing rules significantly more
1204 complex. In particular, the main challenge when developing our type system and its semantic model
1205 came from the interaction between union and concatenation of regions. Viewing types as program
1206 abstractions has been studied theoretically in [15], and done in very different contexts [55].

1207 **Alias and memory analyses:** There are many analyses that assume that a program is well-typed
1208 to derive aliasing information [24, 62]. Using abstract interpretation to combine analyses [18], we
1209 simultaneously prove that programs are well-typed and use this information to compute more
1210 precise alias information, which helps type checking the program. There are flow-sensitive memory
1211 analyses, e.g. [52], for low-level abstraction of memory regions, or shape analyses for low-level
1212 code [26, 39, 41]) that can compute expressive invariants about memory. A type-based analysis like
1213 ours, may be less precise but it is faster and more easily made modular, because the complex heap
1214 invariants are implicitly represented by the flow-insensitive type definitions.

1215 10 CONCLUSION

1216 We have presented a rich non-substructural type system which is *physical* (types represent a
1217 memory layout), *nominal* (pointers to different names do not alias just because they hold the
1218 same content), and *dependent* (it captures complex relations between values and memory layout,
1219 like array sizes). This type system is expressive enough to express many low-level programming
1220 patterns found in C and binary code. We have used this type system to implement a modular abstract
1221 interpreter [19] for low-level code, that performs an automated semantic type checking against
1222 user-provided prototypes. This implies verification of spatial memory safety and has promising
1223 results on challenging benchmarks, allowing us to verify code with a reduced effort.
1224
1225

We have identified three main limitations of our approach. First, our memory invariants cannot represent temporal properties, which prevents us to prove temporal memory safety. Second, our analysis imprecisely handles variable-length arrays and strings. This could be improved by using specific abstractions for arrays [20] and strings [36]. Finally, the writing of type specification still requires some manual work that should be automated in the future. For instance, Lattner’s [40] data structure analysis could be used to automatically refine type specifications by introducing more type names.

DATA-AVAILABILITY STATEMENT

The software that supports §8 is available at <https://zenodo.org/records/10934443> on Zenodo DOI 10.5281/zenodo.10934443. The artifact includes the sources of the analyser CODEX, the set of benchmarks used in §8, and the utilities (makefiles, scripts) to reproduce the results presented in this section.

REFERENCES

- [1] Amal Ahmed. 2004. *Semantics of Types for Mutable State*. Ph.D. Dissertation. Princeton University.
- [2] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. 2010. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.* 32, 3 (2010), 7:1–7:67. <https://doi.org/10.1145/1709093.1709094>
- [3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [4] Andrea Arcangeli, David Woodhouse, and Michel Lespinasse. 2012. *Linux Kernel Red Black Trees*. <https://github.com/torvalds/linux/blob/5133c9e51de41bfa902153888e11add3342ede18/lib/rbtree.c>
- [5] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. 2011. Refinement-Based CFG Reconstruction from Unstructured Programs. In *12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011, Vol. 6538)*. Springer, 54–69. https://doi.org/10.1007/978-3-642-18275-4_6
- [6] Thais Baudon, Gabriel Radanne, and Laure Gonnord. 2023. Bit-Stealing Made Legal: Compilation for Custom Memory Representations of Algebraic Data Types. *Proc. ACM Program. Lang.* 7, ICFP, 813–846. <https://doi.org/10.1145/3607858>
- [7] Richard Bornat. 2000. Proving Pointer Programs in Hoare Logic. In *Mathematics of Program Construction*, Roland Backhouse and José Nuno Oliveira (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 102–126.
- [8] Rodney M Burstall. 1972. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence* 7, 23-50 (1972), 3.
- [9] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. 2018. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning* 61 (2018), 367–422.
- [10] Satish Chandra and Thomas Reps. 1999. Physical type checking for C. *ACM SIGSOFT Software Engineering Notes* 24, 5 (1999), 66–75.
- [11] Bor-Yuh Evan Chang and Xavier Rival. 2013. Modular Construction of Shape-Numeric Analyzers. In *Festschrift for Dave Schmidt (EPTCS, Vol. 129)*, Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff (Eds.). <https://doi.org/10.48550/arXiv.1309.5138>
- [12] Adam Chlipala. 2011. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 234–245.
- [13] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3
- [14] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *ESOP (Lecture Notes in Computer Science, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, 520–535. https://doi.org/10.1007/978-3-540-71316-6_35
- [15] Patrick Cousot. 1997. Types as Abstract Interpretations. In *Conference Record of POPL’97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 316–331. <https://doi.org/10.1145/263699.263744>
- [16] P. Cousot. 2021. *Principles of Abstract Interpretation*. MIT Press. <https://books.google.fr/books?id=CUoQEAAAQBAJ>
- [17] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL*

- 1275 1977). 238–252. <https://doi.org/10.1145/512950.512973>
- 1276 [18] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *6th ACM Symposium*
- 1277 *on Principles of Programming Languages (POPL 1979)*. 269–282. <https://doi.org/10.1145/567752.567778>
- 1278 [19] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Proceedings of the Eleventh International*
- 1279 *Conference on Compiler Construction (CC 2002)*, R.N. Horspool (Ed.). LNCS 2304, Springer, Berlin, Grenoble, France, 159–178.
- 1280 [20] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A Parametric Segmentation Functor for Fully Automatic
- 1281 and Scalable Array Content Analysis. In *Conference Record of the 38th Annual ACM SIGPLAN-SIGACT Symposium on*
- 1282 *Principles of Programming Languages*. ACM Press, New York, Austin, Texas, 105–118.
- 1283 [21] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. 1999. TALx86:
- 1284 A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta,*
- 1285 *GA, USA*. 25–35.
- 1286 [22] Luis Manuel Martins Damas. 1984. *Type Assignment in Programming Languages*. Ph. D. Dissertation. University of
- 1287 Edinburgh.
- 1288 [23] Vincent David, Christophe Aussaguès, Stéphane Louise, Philippe Hilsenkopf, Bertrand Ortolo, and Christophe Hessler.
- 1289 2004. The oasis based qualified display system. In *Fourth American Nuclear Society International Topical Meeting on*
- 1290 *Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT 2004)*, Columbus, Ohio,
- 1291 *USA*. 11.
- 1292 [24] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based Alias Analysis. In *Proceedings of the ACM*
- 1293 *SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98) (PLDI'98)*. ACM, New York,
- 1294 *NY, USA*, 106–117. <https://doi.org/10.1145/277650.277670>
- 1295 [25] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: type and memory error detection using dynamically typed
- 1296 C/C++. In *Proceedings of PLDI*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 181–195. <https://doi.org/10.1145/3192366.3192388>
- 1297 [26] Kamil Dudka, Petr Peringer, and Tomás Vojnar. 2013. Byte-Precise Verification of Low-Level List Manipulation. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings (Lecture Notes*
- 1298 *in Computer Science, Vol. 7935)*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer, 215–237.
- 1299 [27] Adam Dunkels. 2023. *Contiki-OS*. <https://github.com/contiki-ng/contiki-ng/blob/release-4.9/os/lib/list.c>
- 1300 [28] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by
- 1301 Extension. In *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*.
- 1302 *IEEE Computer Society*, 53–60. <https://doi.org/10.1109/SecDev.2018.00015>
- 1303 [29] Free Software Foundation. 2022. *GNU Emacs source repository*. <https://git.savannah.gnu.org/cgiit/emacs.git/tree/lisp.h?h=emacs-28.2>
- 1304 [30] Jacques Garrigue. 2004. Relaxing the Value Restriction. In *Functional and Logic Programming, 7th International*
- 1305 *Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2998)*,
- 1306 Yukiyoshi Kameyama and Peter J. Stuckey (Eds.). Springer, 196–213. https://doi.org/10.1007/978-3-540-24754-8_15
- 1307 [31] Philippe Granger. 1989. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics* 30,
- 1308 3-4 (1989), 165–190. <https://doi.org/10.1080/00207168908803778>
- 1309 [32] Philippe Granger. 1992. Improving the Results of Static Analyses Programs by Local Decreasing Iteration. In *Foundations*
- 1310 *of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992,*
- 1311 *Proceedings (Lecture Notes in Computer Science, Vol. 652)*, R. K. Shyamasundar (Ed.). Springer, 68–79. https://doi.org/10.1007/3-540-56287-7_95
- 1312 [33] Matthew Harren. 2007. *Dependent Types for Assembly Code Safety*. Ph. D. Dissertation. University of California at
- 1313 Berkeley.
- 1314 [34] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A
- 1315 Powerful, Sound, Predictable, Fast Verifier for C and Java. *NASA Formal Methods* 6617 (2011), 41–55.
- 1316 [35] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone:
- 1317 A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.
- 1318 [36] Matthieu Journault, Antoine Miné, and Abdelraouf Oudjaout. 2018. Modular static analysis of string manipulations
- 1319 in C programs. In *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018,*
- 1320 *Proceedings 25*. Springer, 243–262.
- 1321 [37] Stephen Kell. 2016. Dynamically diagnosing type errors in unsafe code. In *Proceedings of OOPSLA*, Eelco Visser and
- 1322 Yannis Smaragdakis (Eds.). ACM, 800–819. <https://doi.org/10.1145/2983990.2983998>
- 1323 [38] Johannes Kinder, Florian Zuleger, and Helmut Veith. 2009. An Abstract Interpretation-Based Framework for Control
- Flow Reconstruction from Binaries. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, Neil D. Jones and Markus Müller-Olm (Eds.). Springer, 214–228. https://doi.org/10.1007/978-3-540-93900-9_19

- [39] Jörg Kreiker, Helmut Seidl, and Vesal Vojdani. 2010. Shape Analysis of Low-Level C with Overlapping Structures. In *VMCAI*. 214–230.
- [40] Chris Lattner. 2005. *Macroscopic Data Structure Analysis and Optimization*. Ph. D. Dissertation. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. See <http://llvm.cs.uiuc.edu>.
- [41] Vincent Laviron, Bor-Yuh Evan Chang, and Xavier Rival. 2010. Separating Shape Graphs. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 387–406. https://doi.org/10.1007/978-3-642-11957-6_21
- [42] Matthieu Lemerre. 2023. SSA Translation Is an Abstract Interpretation. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1895–1924.
- [43] Matthieu Lemerre. 2023. SSA Translation Is an Abstract Interpretation. *Proc. ACM Program. Lang.* 7, POPL, Article 65 (jan 2023), 30 pages. <https://doi.org/10.1145/3571258>
- [44] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. 2022. VIP: verifying real-world C idioms with integer-pointer casts. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–32. <https://doi.org/10.1145/3498681>
- [45] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2014. *The CompCert memory model*. Cambridge University Press, Chapter 32.
- [46] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.* 41, 1 (2008), 1–31. <https://doi.org/10.1007/S10817-008-9099-0>
- [47] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. 2017. Semantic-directed clumping of disjunctive abstract states. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 32–45.
- [48] Huisong Li, Xavier Rival, and Bor-Yuh Evan Chang. 2015. Shape Analysis for Unstructured Sharing. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9291)*, Sandrine Blazy and Thomas P. Jensen (Eds.). Springer, 90–108. https://doi.org/10.1007/978-3-662-48288-9_6
- [49] Aravind Machiry, John H. Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. 2022. C to Checked C by 3C (with appendices). *CoRR* abs/2203.13445 (2022). <https://doi.org/10.48550/arXiv.2203.13445>
- [50] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.* 3, POPL (2019), 67:1–67:32. <https://doi.org/10.1145/3290380>
- [51] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)
- [52] Antoine Miné. 2006. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*. ACM, 54–63. <http://www.di.ens.fr/~mine/publi/article-mine-lctes06.pdf>.
- [53] Antoine Miné. 2006. The octagon abstract domain. *Higher-order and symbolic computation* 19, 1 (2006), 31–100.
- [54] Raphaël Monat and Antoine Miné. 2017. Precise Thread-Modular Abstract Interpretation of Concurrent Programs Using Relational Interference Abstractions. In *Proceedings of VMCAI (LNCS, Vol. 10145)*, Ahmed Bouajjani and David Monniaux (Eds.). Springer, 386–404. https://doi.org/10.1007/978-3-319-52234-0_21
- [55] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPICs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. <https://doi.org/10.4230/LIPICs.ECOOP.2020.17>
- [56] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 527–568.
- [57] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (1999), 527–568. <https://doi.org/10.1145/319301.319345>
- [58] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (Dublin, Ireland) (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 245–258. <https://doi.org/10.1145/1542476.1542504>
- [59] George C. Necula. 1997. Proof-Carrying Code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 106–119. <https://doi.org/10.1145/263699.263712>
- [60] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005),

- 1373 477–526.
- 1374 [61] Olivier Nicole, Matthieu Lemerre, and Xavier Rival. 2022. Lightweight Shape Analysis Based on Physical Types. In *23rd*
- 1375 *International Conference on Verification, Model Checking, and Abstract Interpretation – VMCAI 2022 (Lecture Notes in*
- 1376 *Computer Science, Vol. 13182)*, Bernd Finkbeiner and Thomas Wies (Eds.). Springer, 219–241. [https://doi.org/10.1007/978-](https://doi.org/10.1007/978-3-030-94583-1_11)
- 1377 [62] Jens Palsberg. 2001. Type-based analysis and applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop*
- 1378 *on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, John Field
- 1379 and Gregor Snelting (Eds.). ACM, 20–27. <https://doi.org/10.1145/379605.379635>
- 1380 [63] Marina Polishchuk, Ben Liblit, and Chloë W. Schulze. 2007. Dynamic heap type inference for program understanding
- 1381 and debugging. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*
- 1382 *POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 39–46. <https://doi.org/10.1145/1190216.1190225>
- 1383 [64] Reese T. Prosser. 1959. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *Papers Presented at the*
- 1384 *December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference (Boston, Massachusetts) (IRE-AIEE-ACM '59*
- 1385 *(Eastern))*. ACM, New York, NY, USA, 133–138. <https://doi.org/10.1145/1460299.1460314>
- 1386 [65] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002.*
- 1387 *Proceedings. 17th Annual IEEE Symposium on*. IEEE, 55–74.
- 1388 [66] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. 1995. Supporting dynamic data structures on
- 1389 distributed-memory machines. *ACM Trans. Program. Lang. Syst.* 17 (1995), 233–263. [https://api.semanticscholar.org/](https://api.semanticscholar.org/CorpusID:7251637)
- 1390 [67] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *POPL*. ACM, New York,
- 1391 NY, USA, 131–144. <https://doi.org/10.1145/1706299.1706316>
- 1392 [68] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally
- 1393 with Checked C. In *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the*
- 1394 *European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019,*
- 1395 *Proceedings (Lecture Notes in Computer Science, Vol. 11426)*, Flemming Nielson and David Sands (Eds.). Springer, 76–98.
- 1396 https://doi.org/10.1007/978-3-030-17138-4_4
- 1397 [69] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI '21: 42nd ACM*
- 1398 *SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June*
- 1399 *20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. <https://doi.org/10.1145/3453483.3454036>
- 1400 [70] L. Szekeres, M. Payer, T. Wei, and D. Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security*
- 1401 *and Privacy*. 48–62. <https://doi.org/10.1109/SP.2013.13>
- 1402 [71] Robert D. Tennent and Dan R. Ghica. 2000. Abstract Models of Storage. *High. Order Symb. Comput.* 13, 1/2 (2000),
- 1403 119–129. <https://doi.org/10.1023/A:1010022312623>
- 1404 [72] Mads Tofte. 1990. Type Inference for Polymorphic References. *Inf. Comput.* 89, 1 (1990), 1–34. [https://doi.org/10.1016/](https://doi.org/10.1016/0890-5401(90)90018-D)
- 1405 [73] John Toman, Ren Siqui, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-
- 1406 Sensitive Ownership Refinement Types for Imperative Programs. In *Programming Languages and Systems - 29th*
- 1407 *European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice*
- 1408 *of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*,
- 1409 Peter Müller (Ed.). Springer, 684–714. https://doi.org/10.1007/978-3-030-44914-8_25
- 1410 [74] Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, bytes, and separation logic. In *Proceedings of the 34th*
- 1411 *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19,*
- 1412 *2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 97–108. <https://doi.org/10.1145/1190216.1190234>
- 1413 [75] David Walker. 2005. *Advanced topics in types and programming languages*. The MIT Press Cambridge, Chapter
- 1414 Substructural type systems, 3–44.
- 1415 [76] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP Symb. Comput.* 8, 4 (1995), 343–355.
- 1416 [77] Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation*
- 1417 (1994).
- 1418 [78] Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM*
- 1419 *SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*,
- 1420 Benjamin C. Pierce (Ed.). ACM, 169–180. <https://doi.org/10.1145/507635.507657>
- 1421 [79] Feng Zhou, Jeremy Condit, Zachary R. Anderson, Ilya Bagrak, Robert Ennals, Matthew Harren, George C. Necula,
- and Eric A. Brewer. 2006. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *7th*
- Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 45–60.

1422 A PHYSICAL DEPENDENT TYPES

1423 This section provide additional results on the TYPEDC type-system and the lattice of type-offsets.

1425 A.1 Well-formed type definitions

1426 The set of type definitions Δ induces a derivation relation between types, \prec_Δ defined in Table 2.
 1427 The relation \prec_Δ is well-founded by the Req. 2. The following property of \prec_Δ is used to define the
 1428 join semi-lattice of type-offsets:

1429 **PROPOSITION 1.** *A well-founded relation \prec_Δ induces a DAG over type expressions and a domination*
 1430 *relation \sqsubseteq whose entry is byte.*

1432 **PROOF.** We define a graph $G_{\prec_\Delta} \triangleq (\mathbb{T}, \rightarrow)$ where $\tau_1 \rightarrow \tau_2$ iff $\tau_1 \prec_\Delta \tau_2$. Because \prec_Δ is well-founded,
 1433 there is no loop in this graph so G_{\prec_Δ} is a DAG. The only type with no predecessor by \prec_Δ in Table 2
 1434 is byte.

1435 The domination relation [3, 64] is defined by $\tau \sqsubseteq \tau'$ iff all paths in the DAG of \prec_Δ from the entry
 1436 byte to τ are included in the set of paths from the entry to τ' . \square

1438 A.2 Lattice of type-offsets

1439 The set of closed type-offsets (TO) is $\mathbb{P} \triangleq \mathbb{T} \times \mathbb{N}$. Given an allocation map $\mathfrak{m} \in \mathbb{M}$, the *type-offset*
 1440 *derivation* relation $\prec_{\mathbb{P}, \mathfrak{m}} \subseteq \mathbb{P} \times \mathbb{P}$ is defined in Fig. 5.

1441 Although the relation $\prec_{\mathbb{P}, \mathfrak{m}}$ on \mathbb{P} is linked with the semantics (by \mathfrak{m}), it has also a relation with
 1442 the purely syntactic relation \prec_Δ on types (see Table 2), as stated by the following property which is
 1443 follows from the definitions of these relations:

1444 **LEMMA A.1.** *Let $(\tau, k), (\tau', k') \in \mathbb{P}$ such that $(\tau, k) \prec_{\mathbb{P}, \mathfrak{m}} (\tau', k')$. Then $k \leq k'$ and $\tau \prec_\Delta \tau'$.*

1446 **PROOF.** By induction on the definition of $\prec_{\mathbb{P}, \mathfrak{m}}$ in Fig. 5 using the definition of \prec_Δ in Table 2. \square

1447 The above property has an important corollary: $\prec_{\mathbb{P}, \mathfrak{m}}$ is well-founded since \prec_Δ is well-founded.
 1448 The connection between type-offset and byte tags (ρ, k) is given by the following:

1450 **LEMMA A.2.** *If $(\rho, k) \in \overline{\mathbb{R}}$ is a well-formed byte tag then $\text{path}(\rho, k) \in \mathbb{P}_{\mathfrak{m}}^+$.*

1451 The proof follows from the by induction on the structure of region tags. However, the reverse
 1452 property is not true as shown by the following counterexample:

```
1454 1 def c := a×b; def a := byte;
1455 2 def b := ∃α:(byte with self*self+1==0). byte with self==α;
```

1456 The sequence $\pi = \langle (c, 0), (a, 0), (\text{byte}, 0) \rangle$ is in $\mathbb{P}_{\mathfrak{m}}^+$, but there is no well formed byte tag such that
 1457 $\text{path}(\rho, 0) = \pi$ since $\llbracket c \rrbracket_{\mathfrak{m}} = \llbracket b \rrbracket_{\mathfrak{m}} = \emptyset$ for any \mathfrak{m} . This asymmetry demonstrates the relevance of
 1458 both notions: a byte tag $(\tau/\vec{\rho}, k)$ (and the associated path) gives the position of a byte in a region
 1459 tagged by $\tau/\vec{\rho}$ (and allocated of type τ), while the type offset (τ, k) characterizes all addresses
 1460 having the same type offset, i.e., $\gamma_{\mathbb{P}, \mathfrak{m}}(\tau, k)$ from Fig. 6.

1461 We define below the domination relation between closed type offsets, $\sqsubseteq_{\mathbb{P}, \mathfrak{m}}$. Notice that all
 1462 sequences of type-offset end in $(\text{byte}, 0)$ and the relation $\prec_{\mathbb{P}, \mathfrak{m}}$ is well-founded, so the directed
 1463 graph induced by $\prec_{\mathbb{P}, \mathfrak{m}}$ is acyclic and has as entry $(\text{byte}, 0)$. We denote by $\prec_{\mathbb{P}, \mathfrak{m}}^*$ the reflexive and
 1464 transitive closure of $\prec_{\mathbb{P}, \mathfrak{m}}$.

1465 **Definition A.3 (Domination over type offsets).** For any $(\tau, k), (\tau, k') \in \mathbb{P}_{\mathfrak{m}}$ we say that (τ, k)
 1466 dominates (τ', k') , denoted $(\tau, k) \sqsubseteq_{\mathbb{P}, \mathfrak{m}} (\tau', k')$ iff for any (τ_1, k_1) if $(\tau', k') \prec_{\mathbb{P}, \mathfrak{m}}^* (\tau_1, k_1)$ then
 1467 $(\tau, k) \prec_{\mathbb{P}, \mathfrak{m}}^* (\tau_1, k_1)$.

1469 From the properties of $\prec_{\mathbb{P}, \mathfrak{m}}$ we obtain that:

1471 **THEOREM A.4.** *The domination relation $\sqsubseteq_{\mathbb{P},\mathfrak{m}}$ over $\mathbb{P}_{\mathfrak{m}}$ is a tree relation of root (byte, 0).*

1472 **PROOF.** Clearly (byte, 0) $\prec_{\mathbb{P}_{\mathfrak{m}}}^*(\tau, k)$ for any (τ, k) . Because $\sqsubseteq_{\mathbb{P},\mathfrak{m}}$ is defined using the transitive
1473 closure of $\prec_{\mathbb{P}_{\mathfrak{m}}}$ which is a partial order, then $\sqsubseteq_{\mathbb{P},\mathfrak{m}}$ is a tree relation. \square

1475 The above theorem has as corollary that, for a fixed \mathfrak{m} , we could define a join semi-lattice
1476 $TO_{\mathfrak{m}} = \langle \mathbb{P}_{\mathfrak{m}}, \sqsubseteq_{\mathbb{P},\mathfrak{m}}, \sqcup_{\mathbb{P},\mathfrak{m}} \rangle$ where $(\tau_1, k_1) \sqcup_{\mathbb{P},\mathfrak{m}} (\tau_2, k_2)$ is the least upper bound w.r.t. $\sqsubseteq_{\mathbb{P},\mathfrak{m}}$.

1477 The following theorem states properties of the domination relation with respect to type-offset
1478 concretization into $\mathbb{P}_{\mathfrak{m}}^+$:

1479 **THEOREM A.5.** *Let \mathfrak{m} be an allocation map and $(\tau_1, k_1), (\tau_2, k_2) \in \mathbb{P}_{\mathfrak{m}}$. Then:*

- 1481 (1) $\gamma_{\mathbb{P}^+,\mathfrak{m}}(\tau_1, k_1) \subseteq \gamma_{\mathbb{P}^+,\mathfrak{m}}(\tau_2, k_2)$ iff $(\tau_1, k_1) \sqsubseteq_{\mathbb{P},\mathfrak{m}} (\tau_2, k_2)$, i.e., the domination relation corresponds
1482 to inclusion of type-offset concretizations into well-formed sequences of TO;
- 1483 (2) $\exists(\tau, k) \in \mathbb{P}_{\mathfrak{m}}$ such that $\gamma_{\mathbb{P}^+,\mathfrak{m}}(\tau, k) = \gamma_{\mathbb{P}^+,\mathfrak{m}}(\tau_1, k_1) \cup \gamma_{\mathbb{P}^+,\mathfrak{m}}(\tau_2, k_2)$ iff $(\tau, k) = (\tau_1, k_1) \sqcup_{\mathbb{P},\mathfrak{m}}$
1484 (τ_2, k_2) , i.e., the join operator computes the least common dominator.
- 1485 (3) $\gamma_{\mathbb{P}^+,\mathfrak{m}}(\tau_1, k_1) \cap \gamma_{\mathbb{P}^+,\mathfrak{m}}(\tau_2, k_2) = \emptyset$ iff there is no (τ, k) such that $(\tau, k) \prec_{\mathbb{P}_{\mathfrak{m}}}^*(\tau_1, k_1)$ and
1486 $(\tau, k) \prec_{\mathbb{P}_{\mathfrak{m}}}^*(\tau_2, k_2)$.

1487 **PROOF.** Concretization $\gamma_{\mathbb{P}^+,\mathfrak{m}}$ captures all the paths traversing a type-offset. \square

1488 **THEOREM 5.1.** *Let \mathfrak{m} be an allocation map and $(\tau_1, k_1), (\tau_2, k_2) \in \mathbb{P}_{\mathfrak{m}}$.*

- 1489 (1) If $(\tau_1, k_1) \sqsubseteq_{\mathbb{P},\mathfrak{m}} (\tau_2, k_2)$ then $\gamma_{\mathbb{P},\mathfrak{m}}(\tau_1, k_1) \supseteq \gamma_{\mathbb{P},\mathfrak{m}}(\tau_2, k_2)$, i.e., a dominating type-offset includes
1490 the addresses of the dominated type-offset.
- 1491 (2) If $\gamma_{\mathbb{P}^+,\mathfrak{m}}(\tau_1, k_1) \cap \gamma_{\mathbb{P}^+,\mathfrak{m}}(\tau_2, k_2) = \emptyset$ then $\gamma_{\mathbb{P},\mathfrak{m}}(\tau_1, k_1) \cap \gamma_{\mathbb{P},\mathfrak{m}}(\tau_2, k_2) = \emptyset$, i.e., if there is no path
1492 going through both (τ_1, k_1) and (τ_2, k_2) then they don't share addresses.

1493 **PROOF.** (1) From the fact that $\sqsubseteq_{\mathbb{P},\mathfrak{m}}$ is a domination relation, all the paths from (byte, 0) to
1494 (τ_2, k_2) pass by (τ_1, k_1) . Or $\gamma_{\mathbb{P},\mathfrak{m}}$ capture the addresses of all paths, qed.

1495 (2) Same reasoning as above. \square

1499 B A LOW LEVEL PROGRAMMING LANGUAGE

1500 Our analysis is defined for both C and binary code. For sake of readability, we present the analysis
1501 for a very simple imperative language $\text{WHILE}_{\text{MEM}}$ which includes the most important features of low
1502 level code: assignment, integer and pointer arithmetics, memory allocation, and standard control
1503 flow statements. Features like addressable stack, unstructured control flow and function calls may
1504 be dealt with classical techniques, orthogonal to our approach.

1505 $x \in \mathbb{X}$ program variables $\eta \in \mathcal{N}$ type name
1506 $\ell \in \mathbb{Z}, k \in \mathbb{V}$ integer resp. bit vector constants \diamond bit vector, comparison and logical operators

1507 Expressions $\text{Exp} \ni E ::= k \mid x \mid E \diamond E \mid *_\ell E$
1508 Commands $\text{Cmd} \ni C ::= \text{skip} \mid x := E \mid x := \text{malloc}_\eta(E) \mid *_\ell E := E \mid \text{assume } E$
1509 $\mid C; C \mid \text{while } E \text{ do } C \mid \text{if } E \text{ then } C \text{ else } C$

1510 Fig. 7. Syntax of $\text{WHILE}_{\text{MEM}}$

1514 B.1 Syntax

1515 The syntax of $\text{WHILE}_{\text{MEM}}$ is given by the grammar in Fig. 7. Memory locations are obtained using
1516 dynamic memory allocation and pointer arithmetics. We denote by \mathbb{A} the set of addresses, i.e., values
1517 of memory locations. To simplify the presentation, we consider a little-endian ABI, although the
1518

1520 endianness is a parameter of our analysis. The values \mathbb{V} are bit vectors, i.e., non-negative integers
 1521 represented as fixed size sequences of bytes. The arithmetic, logical and comparison operators
 1522 $\diamond \in \{+, -, \times, /, \&, |, =, \leq, \dots\}$ are extended to bit vector values in \mathbb{V} using the semantic given by the
 1523 ABI in a classic way. Like in C or assembler, we interpret non null expressions as true and null
 1524 expressions as false. Comparison expressions evaluate to 1 when they hold and to 0 otherwise.
 1525 The memory may be read (resp. written) on ℓ bytes at an address given by an expression E using
 1526 the load expression $*_{\ell}E$ (resp. store command $*_{\ell}E := E'$). Due to pointer arithmetics allowed in
 1527 expressions, the grammar of expressions is enough to encode array access $E[E']$ or field access
 1528 $E.E'$. We assume that instances of **malloc** are labeled by type names η defined in the program and
 1529 allocates a number of bytes equal to E times the size of η . The **assume** command always succeeds
 1530 and changes the program's state such that the expression in argument becomes true; it is used to
 1531 insert annotations in the program's abstract state.

1532

1533 B.2 Untyped semantics

1534

Values. The values manipulated by our programs are bit vectors, i.e., non-negative integers
 1535 representable on a given number of bytes. The *set of bit vectors*, denoted by \mathbb{V} , is defined by:

1536

$$1537 \mathbb{V} \triangleq \{(l, v) \mid l \in \mathbb{Z}, v \in [0, 2^{8l} - 1]\}$$

1538

1538 where l is the number of bytes used to represent the value v . The bit vectors are composed using
 1539 *bit vectors concatenation*; given two bit vectors x and y , their concatenation denoted by $x :: y$
 1540 is defined as follows:

1541

$$1542 (l_1, v_1) :: (l_2, v_2) = (l_1 + l_2, v_1 + 2^{8l_1} v_2)$$

1543

1543 The set of *values represented on n bytes* is denoted by \mathbb{V}_n . We denote by \mathcal{W} the *number of bytes*
 1544 *used to store addresses* in our programming language (in most programming languages like C, this
 1545 size would be either 4 or 8). We also use \mathbb{A} to denote the set of address values, i.e., $\mathbb{V}_{\mathcal{W}}$.

1546

1546 *Dynamic semantics.* To define the execution semantics of our programming language, we intro-
 1547 duce the notions of store and heap.

1548

The *variables' store* (or simply *store*) $\sigma \in \Sigma$ maps program variable to their values:

1549

$$1550 \sigma \in \Sigma \triangleq [\mathcal{X} \rightarrow \mathbb{V}]$$

1551

1551 We denote by $\sigma[x]$ the value mapped to x in σ and by $\sigma[x \leftarrow v]$ the update to v of the value
 1552 mapped by σ in x .

1553

The *heap* $h \in \mathbb{H}$ is a partial function that maps addresses to one-byte values:

1554

$$1555 h \in \mathbb{H} \triangleq [\mathbb{A} \rightarrow \mathbb{V}_1]$$

1556

1556 We denote by $h[a..a+l]$ the *load from heap* operation which returns the bit vector of size l obtained
 1557 by concatenation of values $h(a) :: h(a+1) :: \dots :: h(a+l-1)$. The *store to heap* operation at address
 1558 a on l consecutive values with the new value $v \in \mathbb{V}_l$ is denoted by $h[a..a+l \leftarrow v]$.

1559

An untyped state $s \in \mathbb{S}$ is pair of a store and a heap:

1560

$$1561 s = (\sigma, h) \in \mathbb{S} \triangleq (\Sigma \times \mathbb{H})$$

1562

In the following, we denote by $\text{dom}(f)$ the definition domain of function f .

1563

1563 The semantics of expressions is defined by the judgment $s \vdash E \Downarrow v$ saying that in the state $s \in \mathbb{S}$,
 1564 the expression E evaluates to value $v \in \mathbb{V}$. This judgment is defined by the rules in Fig. 8.

1565

1565 The semantics of commands is defined by the judgment $s \vdash C \Rightarrow s'$ meaning that the command
 1566 C transforms the state $s \in \mathbb{S}$ into a new state $s' \in \mathbb{S}$. This judgment is defined by the rules in Fig. 9.

1567

We comment some of these rules.

1568

$$\begin{array}{c}
1569 \\
1570 \\
1571 \\
1572 \\
1573 \\
1574 \\
1575 \\
1576 \\
1577 \\
1578 \\
1579 \\
1580 \\
1581 \\
1582 \\
1583 \\
1584 \\
1585 \\
1586 \\
1587 \\
1588 \\
1589 \\
1590 \\
1591 \\
1592 \\
1593 \\
1594 \\
1595 \\
1596 \\
1597 \\
1598 \\
1599 \\
1600 \\
1601 \\
1602 \\
1603 \\
1604 \\
1605 \\
1606 \\
1607 \\
1608 \\
1609 \\
1610 \\
1611 \\
1612 \\
1613 \\
1614 \\
1615 \\
1616 \\
1617
\end{array}$$

$$\begin{array}{c}
\frac{}{(\sigma, h) \vdash x \Downarrow \sigma[x]} \text{ ENV} \qquad \frac{}{s \vdash k \Downarrow k} \text{ CONST} \\
\frac{s \vdash E \Downarrow a \quad [a, a + \ell] \subseteq \text{dom}(h)}{s \vdash *_\ell E \Downarrow h[a..a + \ell]} \text{ LOAD} \\
\frac{s \vdash E_1 \Downarrow v_1 \quad s \vdash E_2 \Downarrow v_2 \quad v_1, v_2 \in \mathbb{V}_\ell \quad v_2 \neq 0}{s \vdash E_1/E_2 \Downarrow v_1/v_2} \text{ DIV} \\
\frac{s \vdash E_1 \Downarrow v_1 \quad s \vdash E_2 \Downarrow v_2 \quad v_1, v_2 \in \mathbb{V}_\ell \quad \diamond \in \{+, -, \times, \dots\}}{s \vdash E_1 \diamond E_2 \Downarrow v_1 \diamond v_2} \text{ BINOP} \\
\frac{s \vdash E \Downarrow v \quad v \in \mathbb{V}_\ell \quad \triangleright \in \{-, \neg\}}{s \vdash \triangleright E \Downarrow \triangleright v} \text{ UNOP}
\end{array}$$

Fig. 8. Dynamic semantics of expressions in WHILE_{MEM}

$$\begin{array}{c}
1587 \\
1588 \\
1589 \\
1590 \\
1591 \\
1592 \\
1593 \\
1594 \\
1595 \\
1596 \\
1597 \\
1598 \\
1599 \\
1600 \\
1601 \\
1602 \\
1603 \\
1604 \\
1605 \\
1606 \\
1607 \\
1608 \\
1609 \\
1610 \\
1611 \\
1612 \\
1613 \\
1614 \\
1615 \\
1616 \\
1617
\end{array}$$

$$\begin{array}{c}
\text{SKIP} \frac{}{s \vdash \text{skip} \Rightarrow s} \qquad \text{SEQ} \frac{s \vdash C_1 \Rightarrow s_1 \quad s_1 \vdash C_2 \Rightarrow s_2}{s \vdash C_1; C_2 \Rightarrow s_2} \\
\text{ASSIGN} \frac{(\sigma, h) \vdash E \Downarrow v}{(\sigma, h) \vdash x := E \Rightarrow (\sigma[x \leftarrow v], h)} \\
\text{STORE} \frac{(\sigma, h) \vdash E_1 \Downarrow a \in \mathbb{V}_W \quad (\sigma, h) \vdash E_2 \Downarrow v \in \mathbb{V}_I}{(\sigma, h) \vdash *_\ell E_1 := E_2 \Rightarrow (\sigma, h[a..a + \ell \leftarrow v])} \\
\text{ALLOC} \frac{(\sigma, h) \vdash E \Downarrow \ell \quad \ell > 0 \quad [a..a + \ell] \cup \text{dom}(h) = \emptyset \quad v \in \mathbb{V}_\ell}{(\sigma, h) \vdash x := \text{malloc}_\eta(E) \Rightarrow (\sigma[x \leftarrow a], h[a..a + \ell \leftarrow v])} \\
\text{ALLOC0} \frac{(\sigma, h) \vdash E \Downarrow \ell \quad \ell > 0}{(\sigma, h) \vdash x := \text{malloc}_\eta(E) \Rightarrow (\sigma[x \leftarrow 0], h)} \qquad \text{ASSUME} \frac{s \vdash E \Downarrow v \quad v \neq 0}{s \vdash \text{assume } E \Rightarrow s} \\
\text{THEN} \frac{s \vdash \text{assume } E; C_1 \Rightarrow s'}{s \vdash \text{if } E \text{ then } C_1 \text{ else } C_2 \text{ end} \Rightarrow s'} \qquad \text{ELSE} \frac{s \vdash \text{assume } \neg E; C_2 \Rightarrow s'}{s \vdash \text{if } E \text{ then } C_1 \text{ else } C_2 \text{ end} \Rightarrow s'} \\
\text{WHILEEND} \frac{s \vdash E \Downarrow 0}{s \vdash \text{while } E \text{ do } C \text{ done} \Rightarrow s} \\
\text{WHILELOOP} \frac{s \vdash E \Downarrow v \quad v \neq 0 \quad s \vdash C \Rightarrow s_1 \quad s_1 \vdash \text{while } E \text{ do } C \text{ done} \Rightarrow s_2}{s \vdash \text{while } E \text{ do } C \text{ done} \Rightarrow s_2}
\end{array}$$

Fig. 9. Dynamic semantics of statements in WHILE_{MEM}

1618 The *memory write* $*_{\ell}E_1 := E_2$ evaluates E_1 and, if it results in a valid heap address, (i.e. a value in
 1619 \mathbb{A} that is in the domain of h), stores the result of evaluating E_2 at that address, if E_2 evaluates to a
 1620 bit vector size ℓ .

1621 The *memory allocation* $x := \mathbf{malloc}_{\eta}(E)$ makes a non-deterministic choice: either it assigns x to 0,
 1622 or if E evaluates to a positive value ℓ , writes an indeterminate value to a region that was previously
 1623 unmapped in h , and assigns to x the base address of that region.

1624 The *assumption* $\mathbf{assume} E$ selects the state where E evaluates to true (i.e., $\neq 0$).

1625 The *conditional* $\mathbf{if} E \mathbf{then} C_1 \mathbf{else} C_2 \mathbf{end}$ is expressed in terms of assumptions, when E evaluates
 1626 to true (i.e. $\neq 0$), we follow the “then” branch C_1 , otherwise the follow the “else” branch C_2 .

1627 Finally, the semantics of *loops* $\mathbf{while} E \mathbf{do} C \mathbf{done}$ states that if the loop condition E is not longer
 1628 true (or it never was to begin with) then the loop ends. Otherwise, the loop is executed one more
 1629 time. Of course, these rules do not consider loop termination.

1630

1631 B.3 Typed concrete semantics

1632 This is an additional material for §6.

1633

1634 **THEOREM 6.2 (SAFE STORE INSIDE A REGION).** *Let (\mathfrak{m}, h) be a well-typed heap, $a \in \text{dom}(h)$ an*
 1635 *allocated address such that $(\rho, k) \in \overline{\mathfrak{m}}(a)$ and $s = \text{size}(\rho)$. Let $v \in \mathbb{V}_{\ell}$ be a value. Then $h[a..a + \ell \leftarrow v]$*
 1636 *is a safe store iff $h[a - k..a - k + s \leftarrow h[a - k..a] :: v :: h[a + \ell..a - k + s]]$ is a safe store.*

1637 **PROOF.** Follows from $h[a..a + \ell \leftarrow v] = h[a - k..a - k + s \leftarrow h[a - k..a] :: v :: h[a + \ell..a - k + s]]$. \square

1638

1639 **THEOREM 6.2 (NAME MATCHING IMPLIES MONOTONY).** *Let $\mathfrak{m}, \mathfrak{m}' \in \mathbb{A} \rightarrow \mathbb{R}$ such that $\mathfrak{m} \rightsquigarrow \mathfrak{m}'$. Then:*
 1640 $\forall \tau \in \mathbb{T} : (\tau)_{\mathfrak{m}} \subseteq (\tau)_{\mathfrak{m}'}$ and $\llbracket \tau \rrbracket_{\mathfrak{m}} \subseteq \llbracket \tau \rrbracket_{\mathfrak{m}'}$.

1641 **PROOF.** By structural induction on types using the well-founded relation \prec_{Δ} and the definition
 1642 of $(\cdot)_{\mathfrak{m}}$ and $\llbracket \cdot \rrbracket_{\mathfrak{m}}$ in Table 4. \square

1643

1644 C ANALYSIS RULES

1645 We present here the full set of abstract transformers shortly described in §7.

1646

1647 C.1 Rules’s overview and their soundness theorems

1648 As usual in abstract interpretation [18], our abstract transformers (analysis rules) derive from the
 1649 definition of our domain and its concretization. These transformers involve many different opera-
 1650 tions requiring interaction between the different parts of the analysis. To simplify the presentation,
 1651 we describe the abstract transformers as a set of deduction rules over the formulas that our abstract
 1652 domains represent. These formulas are given as judgements below. The rules could be used as a
 1653 proof witness of the type safety of the program.

1654 We sort the rules and the associated judgements in the following classes:

1655

- 1656 • Rules that define the translation of the imperative program constructs into side-effect-free
 1657 symbolic expressions: $\{s_1^{\#}\} E \Downarrow \hat{e} \{s_2^{\#}\}$ for expressions and $\{s_1^{\#}\} C \{s_2^{\#}\}$ for commands;
- 1658 • Rules for inferring typing judgments $s^{\#} \models \hat{e} : \hat{\tau}$ from the contents of the $s^{\#}.\Gamma^{\#}$ component
 1659 of the abstract state, the previous typing judgments, and the relation between symbolic
 1660 expressions;
- 1661 • Rules that infer or adds numerical constraints $s^{\#} \models \hat{e}$ based on conditionals encountered in
 1662 the program or from the inferred typing judgments;
- 1663 • Rules for refining the abstract state $s_1^{\#} \Rightarrow s_2^{\#}$ based on the inferred numerical facts and
 1664 typing judgments;
- 1665 • Rules that infer properties of values read from the heap $\{s_1^{\#}\} *_{\ell} \hat{e}_1 \Downarrow \hat{e}_2 \{s_2^{\#}\}$;

1666

- Rules that check the safety of stores to the heap $\{s_1^\#\} *_{\ell} \hat{e}_1 := \hat{e}_2 \{s_2^\#\}$;
- Rules for joining states $s_1^\# \sqcup s_2^\#$ when there is a merge in the control flow, which involves operations that degrade the precision in the domain (rules LEQ^*), and merge different types and symbolic expressions (rules MATCHING^*).

The following soundness theorem describes the meaning of each kind of rule, and how we prove that each rule is correct. The proofs rely on the concepts described in §5.

THEOREM C.1 (SOUNDNESS OF THE ANALYSIS). *For all $s^\#, s_1^\#, s_2^\# \in \mathbb{S}^\#$; for all $s, s_1, s_2 \in \mathbb{S}$, for all $\hat{e}, \hat{e}_1, \hat{e}_2 \in \hat{\mathbb{E}}$, for all $\hat{\tau}, \hat{\tau}_1, \hat{\tau}_2 \in \hat{\mathbb{T}}$, for all \mathfrak{m} and v :*

Command evaluation *If $\{s_1^\#\} C \{s_2^\#\}$ and $s_1 \in \gamma_{\mathbb{S}^\#}(s_1^\#)$ and $s_1 \vdash C \Rightarrow s_2$, then $s_2 \in \gamma_{\mathbb{S}^\#}(s_2^\#)$.*

Expression evaluation *If $\{s_1^\#\} E \Downarrow \hat{e} \{s_2^\#\}$ and $s \in \gamma_{\mathbb{S}^\#}(s_1^\#)$ and $s \vdash E \Downarrow v$. Let (\mathfrak{m}, v) such that $s \in \gamma_{\mathbb{M}^\#}(s_1^\#)(\mathfrak{m}, v)$. Then $s \in \gamma_{\mathbb{S}^\#}(s_2^\#)$ and $s \in \gamma_{\mathbb{M}^\#}(s_2^\#)(\mathfrak{m}, v)$ and $\gamma_{\hat{\mathbb{E}}}(\hat{e})(v) = v$.*

Reduction *If $s_1^\# \Rightarrow s_2^\#$, then $\gamma_{\mathbb{S}^\#}(s_1^\#) = \gamma_{\mathbb{S}^\#}(s_2^\#)$ and $s_1^\# \sqsubseteq s_2^\#$*

Numerical queries *If $s^\# \models \hat{e}, s \in \gamma_{\mathbb{S}^\#}(s^\#)$ and $s \in \gamma_{\mathbb{M}^\#}(s^\#)(\mathfrak{m}, v)$, then $\gamma_{\hat{\mathbb{E}}}(\hat{e})(v) \neq 0$.*

Typing judgments *If $s^\# \models \hat{e} : \hat{\tau}$ and $s \in \gamma_{\mathbb{S}^\#}(s^\#)$ and $s \in \gamma_{\mathbb{M}^\#}(s^\#)(\mathfrak{m}, v)$, then $\gamma_{\hat{\mathbb{E}}}(\hat{e})(v) \in \gamma_{\hat{\mathbb{T}}}(\hat{\tau})(\mathfrak{m}, v)$.*

Inclusion between symbolic types with offsets *If $s^\# \models (\hat{\tau}_1, \hat{e}_1) \sqsubseteq_{\mathbb{P}}^\# (\hat{\tau}_2, \hat{e}_2)$, then $(\gamma_{\hat{\mathbb{T}}}(\hat{\tau}_1)(\mathfrak{m}, v), \gamma_{\hat{\mathbb{E}}}(\hat{e}_1)(v)) \sqsubseteq_{\mathbb{P}} (\gamma_{\hat{\mathbb{T}}}(\hat{\tau}_2)(\mathfrak{m}, v), \gamma_{\hat{\mathbb{E}}}(\hat{e}_2)(v))$.*

Load operation *If $\{s_1^\#\} *_{\ell} \hat{e}_1 \Downarrow \hat{e}_2 \{s_2^\#\}$, $(\sigma, h) \in \gamma_{\mathbb{S}^\#}(s_1^\#)$, $(\sigma, h) \in \gamma_{\mathbb{M}^\#}(s_1^\#)(\mathfrak{m}, v)$ and $v_1 = \gamma_{\hat{\mathbb{E}}}(\hat{e}_1)(v)$, then $(\sigma, h) \in \gamma_{\mathbb{S}^\#}(s_2^\#)$ and $h[v_1..v_1 + \ell] = \gamma_{\hat{\mathbb{E}}}(\hat{e}_2)(v)$.*

Store operation *If $\{s_1^\#\} *_{\ell} \hat{e}_1 := \hat{e}_2 \{s_2^\#\}$, $(\sigma, h) \in \gamma_{\mathbb{S}^\#}(s_1^\#)$, $(\sigma, h) \in \gamma_{\mathbb{M}^\#}(s_1^\#)(\mathfrak{m}, v)$, $v_1 = \gamma_{\hat{\mathbb{E}}}(\hat{e}_1)(v)$ and $v_2 = \gamma_{\hat{\mathbb{E}}}(\hat{e}_2)(v)$, then $(\sigma, h[v_1..v_1 + \ell] \leftarrow v_2) \in \gamma_{\mathbb{S}^\#}(s_2^\#)$.*

Join $\gamma_{\mathbb{S}^\#}(s_1^\# \sqcup s_2^\#) \supseteq \gamma_{\mathbb{S}^\#}(s_1^\#) \cup \gamma_{\mathbb{S}^\#}(s_2^\#)$.

In the next sections, we present each category of rules and prove their soundness.

C.2 Inclusion rules

We want to define the abstract counterpart of the $\sqsubseteq_{\mathbb{P}}$ operation in lattice TO (see §5). Intuitively, we would want to say that $(\hat{\tau}_1, \hat{e}_1) \sqsubseteq_{\mathbb{P}}^\# (\hat{\tau}_2, \hat{e}_2)$ to mean that given any valuation v , the corresponding (τ_1, k_1) and (τ_2, k_2) that the symbolic type represent are such that $(\tau_1, k_1) \sqsubseteq_{\mathbb{P}}^\# (\tau_2, k_2)$. However, this could be too imprecise, we actually want to consider only the valuations that can be represented by a given state $s^\#$. Hence, our judgments are of the form $s^\# \models (\hat{\tau}_1, \hat{e}_1) \sqsubseteq_{\mathbb{P}}^\# (\hat{\tau}_2, \hat{e}_2)$.

1716

1717

1718

1719

1720

1721

1722

1723

1724

1725

1726

1727

1728

1729

1730

1731

1732

1733

1734

1735

1736

1737

1738

1739

1740

1741

1742

1743

1744

1745

1746

1747

1748

1749

1750

1751

1752

1753

1754

1755

1756

1757

1758

1759

1760

1761

1762

1763

1764

$$\begin{array}{c}
 \text{LEQBYTE} \\
 s^\sharp \models (\hat{\tau}, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\text{byte}[\text{size}(\hat{\tau})], \hat{e}) \\
 \\
 \text{LEQNAMED} \\
 s^\sharp \models (\hat{\eta}, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\Delta(\hat{\eta}), \hat{e}) \\
 \\
 \text{LEQWITH} \\
 s^\sharp \models (\{x : \hat{\tau} \mid \hat{e}_1\}, \hat{e}_2) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}, \hat{e}_2) \\
 \\
 \text{LEQINTER} \\
 \frac{s^\sharp \models (\hat{\tau}_2, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}_3, \hat{e}_3)}{s^\sharp \models (\hat{\tau}_1 \wedge \hat{\tau}_2, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}_3, \hat{e}_3)} \\
 \\
 \text{LEQUNION} \\
 \frac{s^\sharp \models (\hat{\tau}_1, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}, \hat{e}_3) \quad s^\sharp \models (\hat{\tau}_2, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}, \hat{e}_3)}{s^\sharp \models (\hat{\tau}_1 \cup \hat{\tau}_2, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}, \hat{e}_3)} \\
 \\
 \text{LEQPROD} \\
 \frac{s^\sharp \wedge \hat{e} \leq \text{size}(\hat{\tau}_1) \Rightarrow s_1^\sharp \models (\hat{\tau}_1, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}_3, \hat{e}_3) \quad s^\sharp \wedge \hat{e} > \text{size}(\hat{\tau}_1) \Rightarrow s_2^\sharp \models (\hat{\tau}_2, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}_3, \hat{e}_3)}{s^\sharp \models (\hat{\tau}_1 \times \hat{\tau}_2, \hat{e}) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}_3, \hat{e}_3)} \\
 \\
 \text{LEQARRAY} \\
 \frac{s^\sharp \models \text{size}(\hat{\tau}) = \ell \quad s^\sharp \models 0 \leq \hat{e}_2 < \text{size}(\ell * \hat{e}) \quad s^\sharp \models (\hat{\tau}, \hat{e} \bmod \ell) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}_3, \hat{e}_3)}{s^\sharp \models (\hat{\tau}[\hat{e}], \hat{e}_2) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}_3, \hat{e}_3)}
 \end{array}$$

LEQBYTE this rule is used as safe fallback if we fail to find an upper bound when joining types, is notably useful for pointer types.

LEQNAMED, **LEQWITH** directly derive from the definition of these types, which have only one predecessor in the (τ, k) graph.

LEQINTER since $\hat{\tau}_2$ is closer to byte than $\hat{\tau}_1$ in the \prec relation between types, we start from it when trying to find its dominator.

LEQUNION finds a common ancestor between $\hat{\tau}_1$ and $\hat{\tau}_2$. In practice we want the least common ancestor, and this rule corresponds to doing least-common ancestor search in the dominator tree.

LEQPROD in most cases, $\text{size}(\hat{e}_1)$ will be a constant ℓ and one of the condition $\hat{e} \leq \ell$ or $\hat{e} > \ell$ will be \perp , but the rule can handle more complex cases, e.g. when we have a structure containing several integers and the offset inside the structure is not precisely known.

LEQARRAY note the condition $s^\sharp \models 0 \leq \hat{e}_2 < \text{size}(\ell * \hat{e})$, as the rule is incorrect if \hat{e} can be out of bound.

Notice that we don't have a rule for existential types. The existential type must have been instantiated (using rule INSTANTIATE) first.

C.3 Joining states

Intuitively, we want to define the abstract counterpart of the $\sqcup_{\mathbb{P}}$ operation in lattice TO (see §5), that is, given two pairs of symbolic types and offsets $(\hat{\tau}_1, \hat{e}_1)$ and $(\hat{\tau}_2, \hat{e}_2)$, we want to find a pair of symbolic types and offsets $(\hat{\tau}_3, \hat{e}_3)$ such that for any valuation, the concrete type and offset that they represent are related through the $\sqcup_{\mathbb{P}}$ operation. We want to complete this first definition with two elements:

- First, we do not to consider all the possible valuations ν , but only those that correspond to the states s_1^\sharp and s_2^\sharp that we are joining;
- Second, we need to make use of a renaming operation ϕ that will make similar types coincide. For instance, if an address belongs to a set represented by $\text{int}[4] + 1$ in s_1^\sharp , and by $\text{int}[7] + 4$

in s_2^\sharp , then it will be in a set $\text{int}[\alpha_1] + \alpha_2$ in $s_1^\sharp \sqcup s_2^\sharp$, where α_1 and α_2 are fresh symbolic variables that are numerically constrained in $s_1^\sharp \sqcup s_2^\sharp$.

Formally, a renaming function [11] $\phi \in \hat{\mathbb{E}} \times \hat{\mathbb{E}} \rightarrow \mathcal{Q}$ takes a pair of different expression and returns a fresh (or deterministically named [43]) symbol, such that the function is injective (different pairs of expression return a different symbol). This function can be reversed into two substitutions that we call ϕ_1 and ϕ_2 . We extend this function to the case $\phi(\hat{e}, \hat{e}) = \hat{e}$ where the same expression appears twice (and no substitution happens).

The judgment $\vDash_\phi \phi(\hat{\tau}_1, \hat{\tau}_2) = \hat{\tau}$ (which is syntactic) means that ϕ is a suitable renaming function for the renaming of expressions within types. Fig. 10 defines this judgment; we denote by $\phi(\hat{\tau}_1, \hat{\tau}_2) = \hat{\tau}$ the fact that $\hat{\tau}_1$ and $\hat{\tau}_2$ are substituted by $\hat{\tau}$. Formally,

THEOREM C.2. *If $\vDash_\phi \phi(\hat{\tau}_1, \hat{\tau}_2) = \hat{\tau}$, then $\text{subst}(\hat{\tau}_1, \phi_1) = \phi$ and $\text{subst}(\hat{\tau}_2, \phi_2) = \hat{\tau}$.*

$$\begin{array}{l} \text{MATCHINGSAME} \qquad \text{MATCHINGNAME} \\ \vDash_\phi \phi(\hat{\tau}, \hat{\tau}) = \hat{\tau} \qquad \vDash_\phi \phi(\mathbf{n}(\hat{e}_1, \dots, \hat{e}_\ell), \mathbf{n}(\hat{e}_{\ell+1}, \dots, \hat{e}_{2\ell})) = \mathbf{n}(\phi(\hat{e}_1, \hat{e}_{\ell+1}), \dots, \phi(\hat{e}_\ell, \hat{e}_{2\ell})) \end{array}$$

$$\begin{array}{c} \text{MATCHINGWITH} \\ \vDash_\phi \phi(\hat{\tau}_1, \hat{\tau}_2) = \hat{\tau} \\ \hline \vDash_\phi \phi(\{x : \hat{\tau}_1 \mid \hat{e}_1\}, \{x : \hat{\tau}_2 \mid \hat{e}_2\}) = \{x : \hat{\tau} \mid \phi(\hat{e}_1, \hat{e}_2)\} \end{array}$$

$$\begin{array}{c} \text{MATCHINGPROD} \\ \vDash_\phi \phi(\hat{\tau}_{11}, \hat{\tau}_{12}) = \hat{\tau}_1 \quad \vDash_\phi \phi(\hat{\tau}_{21}, \hat{\tau}_{22}) = \hat{\tau}_2 \quad \text{size}(\hat{\tau}_{11}) = \text{size}(\hat{\tau}_{12}) \\ \hline \vDash_\phi \phi(\hat{\tau}_{11} \times \hat{\tau}_{21}, \hat{\tau}_{12} \times \hat{\tau}_{22}) = \hat{\tau}_1 \times \hat{\tau}_2 \end{array}$$

$$\begin{array}{c} \text{MATCHINGARRAY} \\ \vDash_\phi \phi(\hat{\tau}_1, \hat{\tau}_2) = \hat{\tau} \quad \text{size}(\hat{\tau}_1) = \text{size}(\hat{\tau}_2) \\ \hline \phi(\hat{\tau}_1[\hat{e}_1], \hat{\tau}_2[\hat{e}_2]) = \hat{\tau}[\phi(\hat{e}_1, \hat{e}_2)] \end{array}$$

$$\begin{array}{c} \text{MATCHINGUNION} \\ \vDash_\phi \phi(\hat{\tau}_{11}, \hat{\tau}_{12}) = \hat{\tau}_1 \quad \vDash_\phi \phi(\hat{\tau}_{21}, \hat{\tau}_{22}) = \hat{\tau}_2 \\ \hline \vDash_\phi \phi(\hat{\tau}_{11} \cup \hat{\tau}_{21}, \hat{\tau}_{12} \cup \hat{\tau}_{22}) = \hat{\tau}_1 \cup \hat{\tau}_2 \end{array}$$

$$\begin{array}{c} \text{MATCHINGINTER} \\ \vDash_\phi \phi(\hat{\tau}_{11}, \hat{\tau}_{12}) = \hat{\tau}_1 \quad \vDash_\phi \phi(\hat{\tau}_{21}, \hat{\tau}_{22}) = \hat{\tau}_2 \\ \hline \vDash_\phi \phi(\hat{\tau}_{11} \wedge \hat{\tau}_{21}, \hat{\tau}_{12} \wedge \hat{\tau}_{22}) = \hat{\tau}_1 \wedge \hat{\tau}_2 \end{array}$$

$$\begin{array}{c} \text{MATCHINGEXISTS} \\ \vDash_\phi \phi(\hat{\tau}_{11}, \hat{\tau}_{12}) = \hat{\tau}_1 \quad \vDash_\phi \phi(\hat{\tau}_{21}, \hat{\tau}_{22}) = \hat{\tau}_2 \\ \hline \vDash_\phi \phi(\exists \alpha : \hat{\tau}_{11}. \hat{\tau}_{21}, \exists \alpha : \hat{\tau}_{12}. \hat{\tau}_{22}) = \exists \alpha : \hat{\tau}_1. \hat{\tau}_2 \end{array}$$

$$\begin{array}{c} \text{MATCHINGADDR} \\ \vDash_\phi \phi(\hat{\tau}_1, \hat{\tau}_2) = \hat{\tau} \\ \hline \vDash_\phi \phi(\hat{\tau}_1 \star + \hat{e}_1, \hat{\tau}_2 \star + \hat{e}_2) = \hat{\tau} \star + \phi(\hat{e}_1, \hat{e}_2) \end{array}$$

Fig. 10. Renaming function

Most of the rules are simple. Note that:

MATCHINGPROD could be extended when the sizes of the first field do not match, but that would be complicated (and probably not very useful in practice).

MATCHINGADDR could be smarter by making $\phi(\hat{\tau}_1 \star + \hat{e}_1, \hat{\tau}_2 \star + \hat{e}_2)$ match if we can join the pairs $(\hat{\tau}_1, \hat{e}_1)$ and $(\hat{\tau}_2, \hat{e}_2)$. This would make the formalization significantly more complex, in a case where the precision is already very bad (as it represent addresses pointing to a byte[\mathcal{W}] region, with the additional knowledge that it contains a pointer).

With this operator in place, we can (implicitly) define our join operator on abstract type environments by stating that they preserve the following judgment for all the (\hat{e}_1, \hat{e}_2) pairs in $\text{dom}(\phi)$:

$$\begin{array}{c} \text{JOINJUDGMENTS} \\ s_1^\# \models \hat{e}_1 : \hat{\tau}_{11} \star + \hat{e}_{11} \quad s_1^\# \models (\hat{\tau}_{11}, \hat{e}_{11}) \sqsubseteq_{\mathbb{P}}^\# (\hat{\tau}_{111}, \hat{e}_{111}) \\ s_2^\# \models \hat{e}_2 : \hat{\tau}_{22} \star + \hat{e}_{22} \quad s_2^\# \models (\hat{\tau}_{22}, \hat{e}_{22}) \sqsubseteq_{\mathbb{P}}^\# (\hat{\tau}_{222}, \hat{e}_{222}) \\ \hline \vdash_\phi \phi(\hat{\tau}_{111}, \hat{\tau}_{222}) = \hat{\tau} \\ \hline s_1^\# \sqcup s_2^\# \models \phi(\hat{e}_1, \hat{e}_2) : (\hat{\tau} \star + \phi(\hat{e}_{111}, \hat{e}_{222})) \end{array}$$

JOINJUDGMENTS explains how $\Gamma^\#$ is computed: for each pair (\hat{e}_1, \hat{e}_2) of expressions which needs to be joined, we get their respective values $s_1^\#. \Gamma^\#[\hat{e}_1]$ and $s_2^\#. \Gamma^\#[\hat{e}_2]$, climb the $\sqsubseteq_{\mathbb{P}}^\#$ equivalent to the $\sqsubseteq_{\mathbb{P}}$ domination tree on the concrete types to find a matching pair of (symbolic type, symbolic expression) pairs; that we can use as the result (after some renaming using ϕ , which in turns may require new pairs of expression to be added to $\Gamma^\#$, in the case where these expressions could represent some adresses).

We define $\sqcup_{\mathbb{P}}^\phi$ as the smallest $\Gamma^\#$ that can produce these judgments (given a ϕ).

We suppose that $\sqcup_{\mathbb{V}}^\phi$ is given, and \sqcup_{Σ}^ϕ is defined as follows:

$$\sigma_1^\# \sqcup_{\Sigma}^\phi \sigma_2^\# \triangleq \lambda x \in \mathcal{X}. \phi(\sigma_1^\#[x], \sigma_2^\#[x])$$

We can now define the join between two abstract states as:

$$(\sigma_1^\#, \Gamma_1^\#, \nu_1^\#) \sqcup (\sigma_2^\#, \Gamma_2^\#, \nu_2^\#) \triangleq \exists \text{ minimal } \phi : (\sigma_1^\# \sqcup_{\Sigma}^\phi \sigma_2^\#, \Gamma_1^\# \sqcup_{\mathbb{P}}^\phi \Gamma_2^\#, \nu_1^\# \sqcup_{\mathbb{V}}^\phi \nu_2^\#)$$

where *minimal* means that $\text{dom}(\phi)$ should not rename arbitrary pairs of symbolic expressions, only those that are bound to the same variable in both abstract stores, or whose definition is required by rule **JOINJUDGMENTS**.

THEOREM C.3. For all $s_1^\#, s_2^\# \in \mathbb{S}^\#$:

$$\gamma_{\mathbb{S}^\#}(s_1^\# \sqcup s_2^\#) \supseteq \gamma_{\mathbb{S}^\#}(s_1^\#) \cup \gamma_{\mathbb{S}^\#}(s_2^\#)$$

C.4 Rules for compound commands

These rules are usual. Their meaning is given by the following soundness theorem:

1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911

$$\begin{array}{c}
 \text{SKIP} \\
 \frac{}{\{s^\#\} \text{ skip } \{s^\#\}} \\
 \\
 \text{SEQ} \\
 \frac{\{s_0^\#\} C_1 \{s_1^\#\} \quad \{s_1^\#\} C_2 \{s_2^\#\}}{\{s_0^\#\} C_1; C_2 \{s_2^\#\}} \\
 \\
 \text{IF} \\
 \frac{\{s_0^\#\} \text{ assume}(E); C_1 \{s_1^\#\} \quad \{s_0^\#\} \text{ assume}(\neg E); C_2 \{s_2^\#\} \quad s_3^\# = s_1^\# \sqcup^\# s_2^\#}{\{s_0^\#\} \text{ if}(E) C_1 \text{ else } C_2 \{s_3^\#\}} \\
 \\
 \text{WHILE-DONE} \\
 \frac{\{s_0^\#\} \text{ assume}(e); C_1 \{s_1^\#\} \quad s_1^\# \sqsubseteq s_0^\# \quad \{s_1^\#\} \text{ assume}(\neg E) \{s_2^\#\}}{\{s_0^\#\} \text{ while}(E) \text{ do } C_1 \text{ done } \{s_2^\#\}} \\
 \\
 \text{WHILE-AGAIN} \\
 \frac{\{s_0^\#\} \text{ assume}(E); C_1 \{s_1^\#\} \quad \{s_0^\#\} \forall s_1^\# \text{ while}(E) \text{ do } C_1 \text{ done } \{s_2^\#\}}{\{s_0^\#\} \text{ while}(E) \text{ do } C_1 \text{ done } \{s_2^\#\}}
 \end{array}$$

C.5 Rules for basic commands

$$\begin{array}{c}
 \text{ASSIGN} \\
 \frac{\{s_0^\#\} E \Downarrow \hat{e} \{s_1^\#\}}{\{s_0^\#\} x := E \{s_2^\#[\sigma^\# \leftarrow s_1^\#. \sigma^\#[x \leftarrow \hat{e}]]\}} \\
 \\
 \text{ASSUME} \\
 \frac{\{s_0^\#\} E \Downarrow \hat{e} \{s_1^\#\} \quad v_2^\# = s_1^\#. v^\# \wedge (\hat{e} \neq 0) \quad s_1^\#[v^\# \leftarrow v_2^\#] \Rightarrow s_3^\#}{\{s_0^\#\} \text{ assume}(E) \{s_3^\#\}}
 \end{array}$$

ASSIGN evaluates an expression, which returns an updated state which contains new information about this expression, and put in in the abstract store.

ASSUME updates the abstract valuation with the numerical constraints $\hat{e} \neq 0$, propagates this fact, and returns the result.

C.6 Rules for expressions

$$\begin{array}{c}
 \text{VAR} \\
 \{s^\#\} x \Downarrow s^\#. \sigma^\#[x] \{s^\#\} \\
 \\
 \text{LOAD} \\
 \frac{\{s_0^\#\} E \Downarrow \hat{e}_1 \{s_1^\#\} \quad \{s_1^\#\} \star_\ell \hat{e}_1 \Downarrow \hat{e}_2 \{s_2^\#\}}{\{s_0^\#\} \star_\ell E \Downarrow \hat{e}_2 \{s_2^\#\}} \\
 \\
 \text{BINOP} \\
 \frac{\{s_0^\#\} E_1 \Downarrow \hat{e}_1 \{s_1^\#\} \quad \{s_1^\#\} E_2 \Downarrow \hat{e}_2 \{s_2^\#\}}{\{s^\#\} E_1 \diamond E_2 \Downarrow \hat{e}_1 \diamond \hat{e}_2 \{s_2^\#\}} \\
 \\
 \text{CONST} \\
 \{s^\#\} k \Downarrow k \{s^\#\}
 \end{array}$$

These rules just translate program expressions to symbolic expressions. Note that if we want type safety to include absence of runtime errors like division by zero, we can add the hypothesis $s_2^\# \models \hat{e}_2 \neq 0$ to the rule BINOP when the binary operator \diamond is the division.

C.7 The size operator on abstract types

$$\text{size} : \hat{\mathbb{T}} \rightarrow \hat{\mathbb{E}}$$

$$\text{size}(\text{byte}) \triangleq 1$$

$$\text{size}(n(\hat{e}_1, \dots, \hat{e}_\ell)) \triangleq \text{size}(\Delta(n(\hat{e}_1, \dots, \hat{e}_\ell)))$$

$$\text{size}(\eta\star + \hat{e}) \triangleq \mathcal{W}$$

$$\text{size}(\{x : \hat{\tau} \mid \hat{e}\}) \triangleq \text{size}(\hat{\tau})$$

$$\text{size}(\hat{\tau}_1 \times \hat{\tau}_2) \triangleq \text{size}(\hat{\tau}_1) + \text{size}(\hat{\tau}_2)$$

$$\text{size}(\hat{\tau}[\hat{e}]) \triangleq \hat{e} * \text{size}(\hat{\tau})$$

$$\text{size}(\hat{\tau}_1 \cup \hat{\tau}_2) \triangleq \text{size}(\hat{\tau}_1) \text{ if } \text{size}(\hat{\tau}_1) = \text{size}(\hat{\tau}_2)$$

$$\text{size}(\exists\alpha : \hat{\tau}_1. \hat{\tau}_2) \triangleq \text{size}(\hat{\tau}_2) \text{ if } \alpha \notin \text{size}(\hat{\tau}_2)$$

$$\text{size}(\hat{\tau}_1 \wedge \hat{\tau}_2) \triangleq \text{size}(\hat{\tau}_1)$$

We note that this operator fails in two cases:

- when the size of an existential type depends on the existentially-bound variable, or
- when the size of components in a union type is different or cannot be proved equal.

When one of these cases happen, we just return an alarm.

C.8 Rules for load

Rules for load are important as it is the way the analysis has to gain information using the typing invariant.

LOADSIMPLE

$$\frac{s_0^\# \models \hat{e}_1 : (\hat{\tau}\star + \hat{e}_2) \quad s_0^\# \models \text{size}(\hat{\tau}) = \ell \quad s_0^\# \models \hat{e}_2 = 0 \quad \alpha \text{ fresh} \quad s_0^\# \wedge \alpha : \hat{\tau} \Rightarrow s_1^\#}{\{s_0^\#\} \star_\ell \hat{e}_1 \Downarrow \alpha \{s_1^\#\}}$$

LOADLARGER

$$\frac{s_0^\# \models \hat{e}_1 : (\hat{\tau}\star + \hat{e}_2) \quad s_0^\# \models \hat{e}_2 = k \quad s_0^\# \models \text{size}(\hat{\tau}) = \ell_2 \quad \ell_2 \geq k + \ell \quad \{s_0^\#\} \star_{\ell_2} (\hat{e}_1 - k) \Downarrow \hat{e}_3 \{s_1^\#\}}{\{s_0^\#\} \star_{\ell_1} \hat{e}_1 \Downarrow \hat{e}_3[k..k + \ell_1] \{s_1^\#\}}$$

LOADUNIONSPLIT

$$\frac{s_0^\# \models \hat{e} : \hat{\tau}_1 \cup \hat{\tau}_2 \quad s_0^\# \wedge \hat{e} : \hat{\tau}_1 \Rightarrow s_1^\# \quad \{s_1^\#\} \star_\ell \hat{e} \Downarrow \hat{e}_1 \{s_1'^\#\} \quad s_0^\# \wedge \hat{e} : \hat{\tau}_2 \Rightarrow s_2^\# \quad \{s_2^\#\} \star_\ell \hat{e} \Downarrow \hat{e}_2 \{s_2'^\#\}}{\{s_0^\#\} \star_\ell \hat{e} \Downarrow \phi(\hat{e}_1, \hat{e}_2) \{s_1'^\# \sqcup s_2'^\#\}}$$

LOADSIMPLE applies when we load a interval of length ℓ and we have a pointer whose type is $\hat{\tau}\star + 0$ with $\text{size}(\hat{\tau}) = \ell$. In this cas, we create a fresh symbolic variable α , we give it the type $\hat{\tau}$, and we propagate this information (e.g. we propagate with constraints to the numerical domain). Note that when we have more information about the regions where the address may point (e.g. when $\hat{\tau}$ is of the form $\hat{\tau}_1 \wedge \hat{\tau}_2$), we can recover more information about the pointed value.

LOADLARGER applies when the pointer points inside an interval. We can then load the whole interval, and extract the relevant part as the result. It is possible to extend this rule when \hat{e}_2 is not constant (by enumerating the possible values of \hat{e}_2 , loading for each possibility, and joining the resulting states), or by allowing loading of intervals of variable size.

LOADUNIONSPLIT applies when the address is a union type $\hat{\tau}_1 \cup \hat{\tau}_2$. We then fork the analysis according to both cases, assuming that the address is one of $\hat{\tau}_1$ or $\hat{\tau}_2$, propagate this information, continue trying to load with both type, and then join the result.

Note that when the address at which the load is done, \hat{e}_1 , is typed by an existential type, other rules apply like **INSTANTIATE**.

An alternative to **LOADLARGER** is rule **PTRCLIMBADDR LATTICE**, but **LOADLARGER** is more precise if it can apply. However, our implementation does not yet have a good array abstraction and we can only load fixed-size regions, so we use the **PTRCLIMBADDR LATTICE** rule in that case.

Sometimes, the analysis can encounter a situation where none of the above rules apply (e.g. when loading a null pointer). In this case, we raise an alarm in practice.

C.9 Rules for store

STORESIMPLE

$$\frac{s^\# \models \hat{e}_1 : (\hat{\eta} \star + \hat{e}_2) \quad s^\# \models \text{size}(\hat{\tau}) = \ell \quad s^\# \models \hat{e}_2 = 0 \quad \#(\hat{\eta}_3, \hat{e}_3) \text{ s.t. } \hat{\eta}_3 \neq \hat{\eta} \wedge (\hat{\eta}_3, \hat{e}_3) \sqsubseteq_{\mathbb{P}}^\# (\hat{\eta}, \hat{e}_2) \quad \hat{e}_4 : \hat{\eta} \quad (s^\# \wedge (\alpha : \hat{\tau})) \Rightarrow s_1^\#}{\{s^\#\} \star_{\ell} \hat{e}_1 := \hat{e}_4 \{s^\#\}}$$

STORELARGER

$$\frac{s_0^\# \models \hat{e}_1 : (\hat{\tau} \star + \hat{e}_2) \quad s_0^\# \models \hat{e}_2 = k \quad s_0^\# \models \text{size}(\hat{\tau}) = \ell_2 \quad \{s_0^\#\} \star_{\ell_2} (\hat{e}_1 - k) \Downarrow \hat{e}_3 \{s_1^\#\} \quad \{s_0^\#\} \star_{\ell_2} (\hat{e}_1 - k) := \hat{e}_3[0..k] :: \hat{e}_4 :: \hat{e}_3[k + \ell_1.. \ell_2] \{s_1^\#\}}{\{s_0^\#\} \star_{\ell_1} \hat{e}_1 := \hat{e}_4 \{s_1^\#\}}$$

STOREUNIONSPLIT

$$\frac{s_0^\# \models \hat{e} : \hat{\tau}_1 \cup \hat{\tau}_2 \quad s_0^\# \wedge \hat{e} : \hat{\tau}_1 \Rightarrow s_1^\# \quad \{s_1^\#\} \star_{\ell} \hat{e} \Downarrow \hat{e}_2 \{s_1^\#\} \quad s_0^\# \wedge \hat{e} : \hat{\tau}_2 \Rightarrow s_2^\# \quad \{s_2^\#\} \star_{\ell} \hat{e} \Downarrow \hat{e}_2 \{s_2^\#\}}{\{s_0^\#\} \star_{\ell} \hat{e} := \hat{e}_2 \{s_1^\# \sqcup s_2^\#\}}$$

STORESIMPLE The requirement that there is no region deriving from $\hat{\eta}$ can be relaxed, in that it is fine to have derived regions that can contain all the values that $\hat{\eta}$ can.

STORELARGER allows to transform the proof that writing to a part of a region is correct to the proof that writing to the whole region is correct. This is especially important for mild updates, where the tag for the whole interval may change.

STOREUNIONSPLIT does a case split to try to prove that storing the value is correct in every case.

C.10 Other address-manipulating rules

$$\frac{\text{PTRCOMBINEINFO} \quad s^\# \models \hat{e}_1 : \hat{\tau}_1 \star + \hat{e}_2 \quad s^\# \models \hat{e}_1 : \hat{\tau}_2 \star + \hat{e}_2}{s^\# \models \hat{e}_1 : ((\hat{\tau}_1 \wedge \hat{\tau}_2) \star + \hat{e}_2)}$$

$$\frac{\text{PTRCLIMBADDRLATTICE} \quad s^\# \models \hat{e} : (\hat{\tau}_1 \star + \hat{e}_1) \quad s^\# \models (\hat{\tau}_1, \hat{e}_1) \sqsubseteq_{\mathbb{P}}^\# (\hat{\tau}_2, \hat{e}_2) \quad s^\# \models 0 \leq \hat{e}_1 \leq \text{size}(\hat{\tau}_1)}{s^\# \models \hat{e} : (\hat{\tau}_2 \star + \hat{e}_2)}$$

$$\frac{\text{PTRADD} \quad s^\# \models \hat{e} : (\hat{\tau} \star + \hat{e}_1)}{s^\# \models (\hat{e} + \hat{e}_2) : \hat{\tau} \star + (\hat{e}_1 + \hat{e}_2)}$$

PTRCLIMBADDRLATTICE can be used as an alternative to **LOADLARGER** in some cases, and is particularly useful to transforming a load at multiple offsets in an array type into a load at a single offset in an array element. For instance, if we have `def ar := (self:byte | self ≤ 14) [1870]`, and an address \hat{e}_1 of type $ar \star + \hat{e}_2$ with $0 \leq \hat{e}_2 < 1870$, we can infer that $\hat{e}_1 : \{x : \text{byte} \mid x \leq 14\} \star$ to load the value without enumeration. Taking advantage of the introduction of types of the form $\hat{\tau} \star + \hat{e}$, we can infer that the loaded value will be ≤ 14 . **PTRCOMBINEINFO** can be used to combine knowledge about a pointer, to limit the set of regions where the pointer may point. **PTRADD** (and **PTRSUB**, not shown) are the rules used to perform pointer arithmetics.

C.11 Typing rules

$$\begin{array}{cccc} \text{USEGAMMA} & \text{UPCASTNAME} & \text{UPCASTWITH} & \text{PROPAGATEWITH} \\ \frac{s^\#. \Gamma^\#[\hat{e}_2] = (\hat{\tau}, \hat{e})}{s^\# \models \hat{e}_2 : (\hat{\tau} \star + \hat{e})} & \frac{s^\# \models \hat{e} : \hat{\eta}}{s^\# \models \hat{e} : \Delta(\hat{\eta})} & \frac{s^\# \models \hat{e}_1 : \{x : \hat{\tau} \mid \hat{e}_2\}}{s^\# \models \hat{e}_1 : \hat{\tau}} & \frac{s^\# \models \hat{e}_1 : (\{x : \hat{\tau} \mid \hat{e}_2\})}{s^\# \models \hat{e}_2 \neq 0} \end{array}$$

$$\begin{array}{cc} \text{UPCASTPROD1} & \text{UPCASTPROD2} \\ \frac{s^\# \models \hat{e} : (\hat{\tau}_1 \times \hat{\tau}_2) \quad \text{size}(\hat{\tau}_1) = \ell}{s^\# \models \hat{e}[0..\ell] : \hat{\tau}_1} & \frac{s^\# \models \hat{e} : (\hat{\tau}_1 \times \hat{\tau}_2) \quad \text{size}(\hat{\tau}_1) = \ell_1 \quad \text{size}(\hat{\tau}_1 \times \hat{\tau}_2) = \ell}{s^\# \models \hat{e}[\ell_1..\ell] : \hat{\tau}_2} \end{array}$$

$$\begin{array}{cc} \text{UPCASTUNION} & \text{DOWNCASTNAME} \quad \text{DOWNCASTWITH} \\ \frac{s^\# \models \hat{e} : (\hat{\tau}_1 \cup \hat{\tau}_2) \quad s^\# \wedge \hat{e} : \hat{\tau}_2 \Rightarrow \perp}{s^\# \models \hat{e} : \hat{\tau}_1} & \frac{s^\# \models \hat{e} : \Delta(\hat{\eta})}{s^\# \models \hat{e} : \hat{\eta}} \quad \frac{s^\# \models \hat{e}_1 : \hat{\tau} \quad s^\# \models \hat{e}_2 \neq 0}{s^\# \models \hat{e}_1 : (\{x : \hat{\tau} \mid \hat{e}_2\})} \end{array}$$

$$\begin{array}{ccc} \text{DOWNCASTPROD} & \text{DOWNCASTUNION} & \text{DOWNCASTEXISTS} \\ \frac{s^\# \models \hat{e}_1 : \hat{\tau}_1 \quad s^\# \models \hat{e}_2 : \hat{\tau}_2}{s^\# \models (\hat{e}_1 :: \hat{e}_2) : (\hat{\tau}_1 \times \hat{\tau}_2)} & \frac{s^\# \models \hat{e} : \hat{\tau}_1}{s^\# \models \hat{e} : (\hat{\tau}_1 \cup \hat{\tau}_2)} & \frac{s^\# \models \hat{e}_1 : \hat{\tau}_1 \quad s^\# \models \hat{e}_2 : \text{subst}(\hat{\tau}_2, [\alpha \rightarrow \hat{e}_1])}{s^\# \models \hat{e}_2 : (\exists \alpha : \hat{\tau}_1. \hat{\tau}_2)} \end{array}$$

$$\begin{array}{cc} \text{TYPEQ} & \text{ALLBYTE} \\ \frac{s^\# \models \hat{e}_1 : \hat{\tau} \quad s^\# \models \hat{e}_1 = \hat{e}_2}{s^\# \models \hat{e}_2 : \hat{\tau}} & \frac{\text{size}(\hat{e}) = \ell}{s^\# \models \hat{e} : \text{byte}^\ell} \end{array}$$

We call upcast the rules that go from a type to its subterm type, and downcast the rules that go in the opposite direction (note that in general our upcast rules do not require side conditions while downcast rules do; however it is the opposite for union types). In general, the **UPCAST** rules

are used whenever necessary, in particular to find the target of addresses in a `LOAD` operation, or to join pointer types when two states are merged. The `DOWNCAST` rules are used for the `STORE` operation, where we try to check that a given value can be casted into a given type for the store operation to be safe.

Most of the rules are straightforward. Note however that:

- Our rules for `UPCASTPROD1` and `UPCASTPROD2` currently have fixed-size condition on some of the types (that could be lifted if the symbolic expressions can represent variable-sized bit vectors).
- The `DOWNCASTEXISTS` rule is difficult to apply, as it corresponds to quantifier elimination, for which we need to find a correct α . We have implemented a simple version which works when $\hat{\tau}_2$ contains a refinement constrain of the form `self = α` , as when such a rule exist, finding the matching α to be substituted is simple.
- There are different cases of propagation of numerical information to the type domain:
 - `UPCASTUNION` eliminates of impossible members in a union type;
 - `DOWNCASTWITH` allows proving that a with constraint holds for a symbolic expression;
 - `TYPEQ` allows to combine equality information to type information.
- The most important numerical information obtained from types is obtained by the rule `PROPAGATEWITH`.
- There is no rule `UPCASTEXISTS`: we always use it in the rule `INSTANTIATE`, which also propagates information learned during the instantiation.
- `ALLBYTE` allows to combine equality information to type information, and can be used as the starting point for applying `DOWNCAST` rules.

C.12 Reduction rules

Reduction [18, 32] (also called constraint propagation) is an operation which improves the abstract element without changing the concretization.

$$\begin{array}{c}
 \text{INSTANTIATE} \\
 \frac{s_0^\# \models \hat{e} : (\exists \alpha : \hat{\tau}_1. \hat{\tau}_2) \quad \alpha_1 \text{ fresh} \quad s_0^\# \wedge \alpha_1 : \hat{\tau}_1 \Rightarrow s_1^\# \quad s_1^\# \wedge \hat{e} : \text{subst}(\hat{\tau}_2, [\alpha \rightarrow \alpha_1]) \Rightarrow s_2^\#}{s_0^\# \Rightarrow s_2^\#} \quad \text{NUMREDUC1} \quad \frac{s^\# \models \hat{e} \neq 0 \quad s^\#.v^\# \wedge (\hat{e} \neq 0) \Rightarrow v_1^\#}{s^\# \Rightarrow s^\#[v^\# \leftarrow v_1^\#]} \\
 \\
 \frac{\text{NUMREDUC2} \quad s^\#.v^\# \Rightarrow v_1^\#}{s^\# \Rightarrow s^\#[v^\# \leftarrow v_1^\#]} \quad \frac{\text{TYPREDEC} \quad s^\# \models \hat{e} : \hat{\eta}_1 \star + \hat{e}_1 \quad s^\# \models (\hat{\eta}_1, \hat{e}_1) \sqsubseteq_{\mathbb{P}}^\# s^\#. \Gamma^\# [\hat{e}] \vee \hat{e} \notin \text{dom}(s^\#. \Gamma^\#)}{s^\# \Rightarrow s^\#[\Gamma^\# \leftarrow s^\#. \Gamma^\# [\hat{e} \leftarrow (\hat{\eta}_1, \hat{e}_1)]]}
 \end{array}$$

INSTANTIATE this rule is used to open an existentially-bound type by instantiating its parameter α into a fresh variable α_1 . We try to apply this rule as soon as possible; in particular whenever we load a field in a structure, then we load all the fields of the structure and we instantiate all the existentially-bound variables.

NUMREDUC1 propagates an inferred fact in the numerical abstract domain.

NUMREDUC2 performs constraint propagation in the numerical abstract domain (e.g. performs transitive closure in the octagon [53] abstract domain), and updates the main domain accordingly.

TYPREDEC saves typing judgments that have been inferred in the abstract type environment $\Gamma^\#$. To ensure that we are performing reduction, we only save results that are more precise than what already existed in $\Gamma^\#$. The most important thing to note in this rule is that we

2108 don't save arbitrary $(\hat{\tau}_1, \hat{e}_1)$ in Γ^\sharp , only rules of the form $(\hat{\eta}_1, \hat{e}_1)$. The reason is that arbitrary
 2109 $\hat{e} : \hat{\tau}_1 \star + \hat{e}_1$ rules may not be preserved when the store operation performs a mild update, as
 2110 we discussed in §6; so here we limit the contents of Γ^\sharp to only contain typing judgments
 2111 that are preserved by store. This can be relaxed in two ways; first, because judgments of
 2112 the form $\hat{e} : (\hat{\tau}_1 \wedge \hat{\tau}_2) \star + \hat{e}_1$ will also be preserved when there is a mild update, so they can
 2113 be saved too. Second, it is possible to save arbitrary judgments $\hat{e} : \hat{\tau}_1 \star + \hat{e}_1$ in Γ^\sharp , provided
 2114 that the ones that are not preserved are removed when they may be affected by a store
 2115 operation. This would allow the analysis to be more precise (at the sake of more complexity
 2116 of the presentation).
 2117

2118 D TRUE ALARMS AND PROBLEMS FOUND WHILE ANALYZING THE BENCHMARKS

2119 The following summarizes the alarms that we found during the analysis of the benchmarks in §8.

- 2120 • **Olden/bh** In the function `maketree`, if the mass of all the bodies are not massive, the
 2121 function `hackcofm` is called with a null pointer, which makes it fail.
- 2122 • **Olden/mst** The code never checks if pointers returned by `malloc` are null.
- 2123 • **Shapes/javl** The original code was adapted for use as a benchmark in Li et al. [47], and the
 2124 macros were changed (notably to remove some `... while(0)` constructs). But nested
 2125 inclusion of these new macros introduced variable capture issues on the `ndir` variable that
 2126 led to alarms during the analysis of function `jsw_avlinsert`.
- 2127 • **Shapes/graph** The function `node_add` allocates some memory but does not write the
 2128 pointer anywhere, leading to a memory leak.

2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156