A Dependent Nominal Physical Type System for Static Analysis of Memory in Low Level Code (with appendices)

JULIEN SIMONNET, University Paris-Saclay, CEA, List, France
MATTHIEU LEMERRE, University Paris-Saclay, CEA, List, France
MIHAELA SIGHIREANU, University Paris-Saclay, ENS Paris-Saclay, 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.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Theory of computation** → **Type theory**; **Program analysis**.

Additional Key Words and Phrases: Abstract interpretation, Dependent types, Spatial memory safety, Type checking, Typed C

ACM Reference Format:
Julien Simonnet, Matthieu Lemerre, and Mihaela Sighireanu. 2024. A Dependent Nominal Physical Type System for Static Analysis of Memory in Low Level Code (with appendices). *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 272 (October 2024), 45 pages. https://doi.org/10.1145/3689712

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

Authors' Contact Information: Julien Simonnet, University Paris-Saclay, CEA, List, Palaiseau, France, julien.simonnet@cea.fr; Matthieu Lemerre, University Paris-Saclay, CEA, List, Palaiseau, France, matthieu.lemerre@cea.fr; Mihaela Sighireanu, University Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, Gif-sur-Yvette, France, mihaela.sighireanu@ens-paris-saclay.fr.

ACM 2475-1421/2024/10-ART272
https://doi.org/10.1145/3689712

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, 78].

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 [14] 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 *strong update* [75]). However, these solutions require complex reasoning on the program state. For example, they may require to track all the aliases to the changed memory region, which is difficult when the pattern used for sharing references is complex, like in garbage-collected programs or operating systems code. Another illustration of this difficulty is the need to abandon structural properties [77] when aliasing is controlled using a type system (e.g., using ownership types).

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

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

By low-level programs we mean programs implemented in low-level languages like C or machine code. The memory invariants in those programs must use low-level concepts, i.e., at the level of byte representation of values. Consequently, expressing these invariants using types requires expressive features, in particular, *dependent types*. For instance, low-level arrays are encoded as a pointer (the array base) to a region whose size is given in another integer (the array size), which is a classic example of the use of dependent types. Furthermore, the bytes holding the array base may not adjoin the array size, illustrating the need for *non-local invariants* [33]. Another important feature is the frequent usage of *interior pointers* (pointing inside a record), and of functions operating over interior pointers of different datatypes as soon as part of the data layout is compatible (this is called *physical typing* in [11, 63]). Finally, low-level languages heavily make use of idioms allowing for efficient usage of the computer resources, such as *union* types, or taking advantage of pointer alignment to reuse the least significant bits in pointer values (*bit-stealing* [7]).

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

Nominal systems of regions have already been used to verify imperative programs, e.g. [8, 15, 28, 35, 60, 67]. However, these systems are generally *flat*, with no embedding or combination between memory regions. The main original feature of our type system is its ability to **combine regions by concatenation (record and array types), union (union and existential types), or sharing of parameters (global properties), in addition to refining regions (using refinement and intersection types)**. This rich composition of memory regions is the key to handling the low-level programming patterns previously described, as we demonstrate in §2, without having to resort to ad-hoc types, tailored for the specific application, and typing rules as is often done [60, 69]. Furthermore, the combination between region composition and explicit naming brings up several interesting properties. First, reasoning about points-to and aliasing properties (inclusion or disjunction of sets of addresses) reduces to reasoning about the graph expressing how definitions of type names (η) derive from each other (see §4). Second, a store can change the type of the value contained in a memory region, provided a checkable condition that all region tags are compatible with this change (see §6). We call this a *mild update* semantic of store because it combines the benefits of strong update (the ability to change the type of a memory cell) and of weak update (there is no need to track aliases) semantics.

This type system could be used as a typed assembly language [3, 55, 57, 80] and carried as a proof [59] that the program is type-safe. The proof may be produced by a type-preserving compiler for a type-safe language and checked using syntactic proof rules. However, our goal is to prove low-level programs to be type-safe *automatically, without modification, even if the program was written in an unsafe language like C or binary code.* Specifically, **we want to prove type-safe programs written in type-unsafe languages like C, or programs transformed (using any tool) into type-unsafe machine code, without any modification, given user-provided interface (type definitions and function prototypes) using the types of our type system**. The absence of code modification is important as modifying the code or executable may significantly worsen its performance [72], and requires significant efforts which hinders adoption.

Relying on particular syntactic constructs to achieve this goal, as done in methods based on syntactic type checking [15, 35, 60, 68], would defeat this goal of verifying the code unmodified, and would probably make verification of machine code impossible. To achieve our goal and reach a sufficient level of precision, **we make our type system and type checking algorithm semantic using abstract interpretation** [18] (i.e., we formalize our type system semantically [3, 22, 50, 74]), and implement it by a standard flow-sensitive static analysis. This semantic type-checking design brings us at least three benefits. (i) It allows us to (try to) type check programs despite intrinsic undecidability, with good results in practice. (ii) It provides us with a systematic method [19] to state and prove the soundness of our typing rules, which corresponds to the static analysis operations. (iii) It gives us modular extensibility, as the type checking can be seamlessly combined with other numerical, memory, or control-flow analyses to improve its precision, overcoming the syntactic limitations of purely syntactic type-checking methods. For instance, we can extend our analysis with points-to predicates [61] that provides additional relations between an address and its contents, and simultaneously makes use of the aliasing information provided by our types. We can also use in parallel abstract domains for proving control safety by control-flow reconstruction [6, 38], which allows us to remove from our type system the control-flow properties.

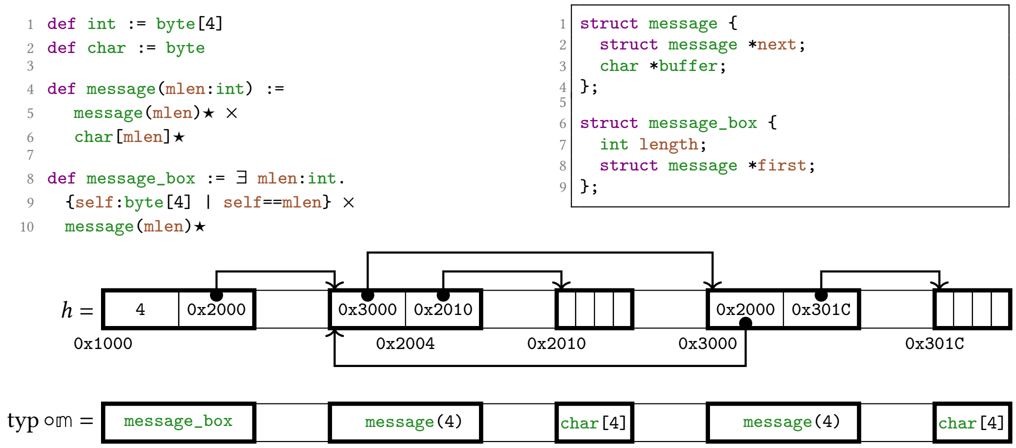
This focus on automation led us to choose a structural type system which does not track all the references to a memory region (we can forget about some references using the Weakening structural property). Indeed, the absence of reference tracking reduces (i) the annotation burden by avoiding ownership or borrowing annotations, like in Rust, and (2) the risk of failure to type-check caused by imprecision in tracking all the references. However, **this choice prevents us to verify programs that perform arbitrary type-changing writes**. This is less limiting than one may think as (i) most writes are not type-changing in C programs and (ii) some type-changing writes may be type-checked without sub-structural rules using an expressive structural type system like TypedC (see example page 9). Still, **the main limitation of our type-system is that it cannot express nor prove temporal properties** (like temporal memory safety or absence of data races) **or any property that requires a temporal invariant on the heap** (like type-changing writes that are not in the class of mild updates), because such properties require a substructural type system. However, our analysis is based on abstract interpretation, a method which provides a framework for composition of analyses. Therefore, by composing our analysis with some dealing with temporal properties (e.g., shape analysis or Monat and Miné [53] method for computing modular effects of threads), we would 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 [1], 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 [8, 9] as well as on many type systems for C [15, 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.*



```
def int := byte[4]
def char := byte

def message(mlen:int) :=
   message(mlen)★ ×
   char[mlen]★

def message_box := ∃ mlen:int.
  {self:byte[4] | self==mlen} ×
  message(mlen)★
```

```
struct message {
  struct message *next;
  char *buffer;
};

struct message_box {
  int length;
  struct message *first;
};
```

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

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., [15, 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 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 shape[1], an over-approximation of the circularity.

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

```
def tag(y:int) := {self:byte[4] | self==y};

def buffer := ∃ sz:int.
  tag(12) ×
  {self:byte[4] | self==sz} ×
  Lisp_Object ×
  char[sz]★

def vector := ∃ len:int.
  tag(0) ×
  {self:byte[4] | self==len} ×
  Lisp_Object[len]

def vectorlike_p :=
  buffer★ ∪ vector★
```

```
struct buffer {
  int header;
  int size;
  Lisp_Object name;
  char *contents;
};

struct vector {
  int header;
  int size;
  Lisp_Object array[];
};

typedef union {
  struct buffer *buf;
  struct vector *vec;
} vectorlike_p;
```

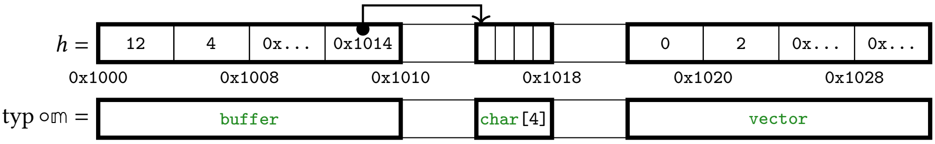


Fig. 2. Specification (left) for the data structure (right) used by the Emacs-Lisp interpreter for vector-like values and a possible memory layout (bottom)

To 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 ∘𝕞, is based on what we call an *allocation map* 𝕞, mapping the same addresses as h to the region tags specifying the values that the memory cells at these addresses should contain [76]. 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 𝕞 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 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 [5] 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

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

```
def rbcolor := {self:byte[4] | (self==RED ∨ self==BLACK)}

def myrb_node_p :=
      {self:(mytype★ +4) | self%4==0}
    ∪ {self:byte[4] | self==0}

def myrb_node :=
    myrb_node_color × myrb_node_p × myrb_node_p

def myrb_node_color :=
    ∃α :myrb_node_p,γ:rbcolor. {self:byte[4] | self==α|γ}

def mytype := char[16]★ × myrb_node
```

```
#define RED 0
#define BLACK 1

struct rb_node {
  int rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};

struct mytype {
  char *key;
  struct rb_node node;
};
```

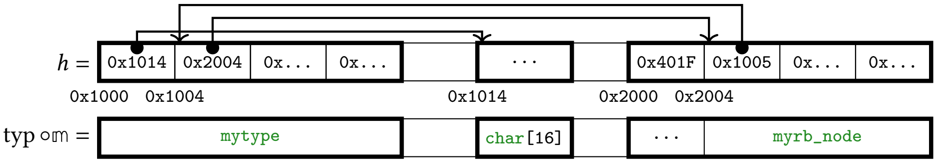


Fig. 3. Specification (left) of the C types for red black trees used in the Linux kernel and their usage (right) with a possible memory layout (at bottom)

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. 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 the right of Fig. 4 defines a binary tree where each node maintains a reference to its parent (root's parent is null). The three functions are memory safe because they rely on the following memory invariant (i) they are called with non null references to nodes, (ii) a node is either a leaf node (i.e., where both children are null) or an interior node (inode) where both children are non null (i.e., of type node★), and (iii) the parent node, if it exists, is an interior node (inode?). These invariants are specified to the 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. This specification is enough to type-check and thus prove memory safety of the next_right function because if its parent is not null, it is an interior node with a non null right child. Type-checking extend and to_leaf requires to deal with type-changing updates.

For instance, 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★. A *weak update* semantics which requires that a write in a memory region does not change the region's type, does not suffice to type check this update: we need to change the case of the union and to reflect this in the type. A *strong update* semantics could type check the program, but it would require to check that n is the only pointer to the updated memory region or that any aliasing pointer would also have its type changed to inode★. Checking such conditions is challenging when nodes can be arbitrarily shared, as it requires additional annotations about the number of references to an object (e.g., ownership) or a complex shape analysis.

One of our contributions is identifying a third way to allow safe updates changing the case of a union type without changing the types of existing values or requiring to track all the pointers to the union. We call this new semantic a *mild update*. Basically, type-changing writes are allowed provided that existing pointers will preserve theirs type. In the case of extend, all values of type node★ still have type node★ after the call, and all values of type inode★ still have type inode★: all typing judgements about pointers are preserved. The set of values of type inode★ evolves though: there are new values of type inode★, as the variable n holds now a inode★ (which makes safe the assignments on line 10). Indeed, mild update requires the set of values in pointer types to grow. Conversely, the to_leaf function transforms an interior node to a leaf node: the assignments on line 13 remove the address in n from the set of values of type inode★. Intuitively, this code is not type safe because there may exist some pointer of type inode★ that would alias with n. Consequently, the invariant that this pointer points to an inode is no longer true after the function's call, so this function does not type-check. The difference between to_leaf and extend is that pointers to interior nodes are possible (because we have a type name, inode, for that) while pointers to leaf nodes are not (i.e., it is an anonymous type expression in node). Thus, weak update permits to change a node from leaf to interior, but forbids to change from interior to leaf node. If we wanted to_leaf to type check, we could remove the inode name and inline its type expression in the definition of node, then convert pointers to inode to pointers to node. This will allow arbitrary changes between the cases of the node type. However, the new invariant is no longer strong enough to type-check next_right, and the analysis will raise an alarm. A pragmatic solution to this problem would be to add an assertion that n->parent->right is non null. This example illustrates the main limitation of our work: to prove both next_right and to_leaf without changing the code, we would need a more complex substructural type system or an advanced shape analysis, capable to deal with strong updates.

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.

```
nullptr ≜ {p:byte[𝒲] | p==0}
inode? ≜ inode★ ∪ nullptr
def int := byte[4]
def inode := inode?
           × int
           × node★
           × node★
def node := ( inode?
             × int
             × nullptr
             × nullptr)
           ∪ inode
inode★ extend(node★,node★,node★);
node★ to_leaf(node★);
node★ next_right(node★);
```

```
typedef struct node_s node;
struct node_s {
  node *parent;
  int val;
  node *right;
  node *left;
};
node* extend(node *n, node *l, node *r) {
  n->left = l; n->right = r;
  l->parent = r->parent = n; return n;
}
node* to_leaf(node *n) {
  n->left = n->right = 0; return n;
}
node* next_right(node *n) {
  return (n->parent!=0)?n->parent->right:n;
}
```

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

Table 1. Physical dependent types of TypedC

symbolic variables: α, self $\in \mathbb{A}$	integer constants: $\ell \in \mathbb{Z}$, bit-vector constants: $k \in \mathbb{V}$
type identifiers: n $\in \mathcal{N}$	operators: $\diamond$ (bit-vector, comparison and logical)
$\Delta \ni$ def $\mathsf{n}(\alpha_1 : \tau_1, \ldots, \alpha_\ell : \tau_\ell)$:= τ	type name definition with $fv(\tau) \subseteq \{\alpha_1, \ldots, \alpha_\ell\}$
Type names η ::= byte $\mid \mathsf{n}(e_1, \ldots, e_\ell)$	
Types $\mathbb{T} \ni \tau ::= \eta \mid \eta\star + e \mid \{\mathtt{self} : \tau \mid e\} \mid \tau_1 \times \tau_2 \mid \tau[e] \mid \tau_1 \cup \tau_2 \mid \exists\alpha : \tau_1 . \tau_2 \mid \tau_1 \wedge \tau_2$	
Terms $\mathbb{E} \ni e ::= k \mid \alpha \mid e_1 \diamond e_2 \mid e_1 :: e_2 \mid e[k_1..k_2]$	

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 $\Delta(\mathsf{n}(e_1, \ldots, e_\ell))$ the type expression obtained by substituting the parameters in the definition of n by the values of terms $e_1, \ldots, e_\ell$ 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 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 η?, are encoded using union and refined type expressions (see Fig. 3 and

Table 2. Orderings $\prec_\Delta$ and $\prec_\Delta^\exists$ between type expressions.

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

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 "$\{\mathtt{self} : \tau \mid e\}$" 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., $\{\mathtt{self} : \mathtt{byte} \mid 1 == 0\}$ 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 are 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]$\star$ (e.g., in type buffer from Fig. 2) a pointer to an array of sz characters, instead of arr_char(sz)$\star$ 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 $\eta\star \cup \{\mathtt{self} : \mathtt{byte}[\mathcal{W}] \mid \mathtt{self} == 0\}$, 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 union[3]. 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 "$\tau_1 \mathbin{/\!\!\backslash} \tau_2$" 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 $\mathbb{V}$, 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 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, def η := τ 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

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

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 $\prec_\Delta$, which is formally defined in Table 2. The relation $\prec_\Delta$ 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 $\mathtt{byte}[\mathcal{W}] \prec_\Delta \eta\star + e$ (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 $\tau_2 \prec_\Delta \tau_1 \mathbin{/\!\!\!\wedge} \tau_2$ 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 $\prec_\Delta$ to $\tau_1 \mathbin{/\!\!\!\wedge} \tau_2$:

Requirement 1 (Well-formed intersection types). *Any type expression $\tau_1 \mathbin{/\!\!\!\wedge} \tau_2$ 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 $\prec_\Delta$ with the relation $\prec^\exists_\Delta$ defined by $\tau_1 \prec^\exists_\Delta \exists\alpha : \tau_1.\, \tau_2$ and we ask the following:

Requirement 2 (Well-formed Δ). *The relation $\prec_\Delta \cup \prec^\exists_\Delta$ induced by Δ is well-founded.*

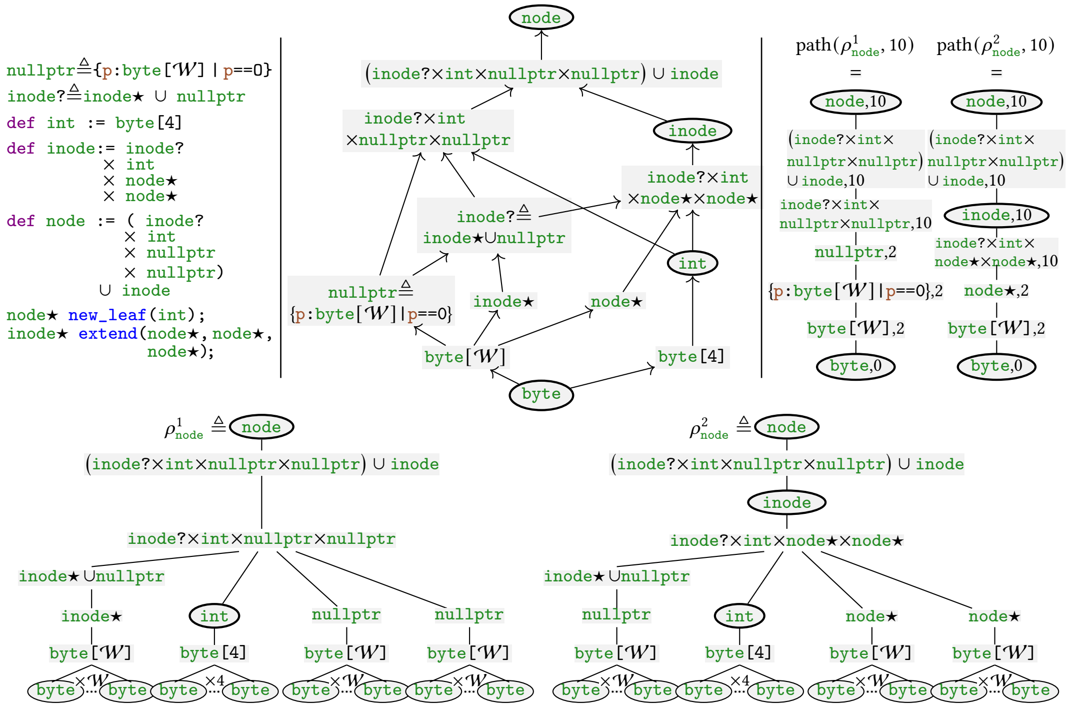
The above requirement implies in particular the following property:

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

An example of such a graph is given in Fig. 5. The DAG induced by $\prec_\Delta$ (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`.

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 $\tau*$ represents all the start addresses of regions containing a value of type τ. Our type system is *nominal*: pointer types $\eta\star$ 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.



```
nullptr≜{p:byte[𝒲] | p==0}
inode?≜inode★ ∪ nullptr
def int := byte[4]
def inode:= inode?
          × int
          × node★
          × node★
def node := ( inode?
            × int
            × nullptr
            × nullptr)
          ∪ inode
node★ new_leaf(int);
inode★ extend(node★, node★,
              node★);
```

node
(inode?×int×nullptr×nullptr) ∪ inode
inode?×int ×nullptr×nullptr
inode
inode?×int ×node★×node★
inode?≜ inode★∪nullptr
int
nullptr≜ {p:byte[𝒲] | p==0}
inode★
node★
byte[𝒲]
byte[4]
byte
path($\rho^1_{\texttt{node}}$, 10) =
path($\rho^2_{\texttt{node}}$, 10) =
node,10
(inode?×int× nullptr×nullptr) ∪ inode,10
inode?×int× nullptr×nullptr,10
nullptr,2
{p:byte[𝒲] | p==0},2
byte[𝒲],2
byte,0
inode,10
inode?×int× node★×node★,10
node★,2
$\rho^1_{\texttt{node}}$ ≜ node
$\rho^2_{\texttt{node}}$ ≜ node
inode?×int×nullptr×nullptr
inode?×int×node★×node★
inode★ ∪nullptr
×𝒲
×4
byte ... byte

Fig. 5. Full specification (top left), DAG representation of $\prec_\Delta$ (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.

4.1 Memory Model

We first introduce some notations. We denote by $\langle\rangle$ the empty sequence, $\langle\sigma_1, \ldots, \sigma_n\rangle$ is a sequence of n elements, $\vec{\sigma}$ is a sequence of $|\vec{\sigma}|$ elements with $\vec{\sigma}[i]$ the ith element of $\vec{\sigma}$. Concatenation $\vec{\sigma} \cdot \vec{\sigma}'$ and the last element $\mathrm{last}(\vec{\sigma})$ 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. $\mathrm{dom}(f)$ and $\mathrm{img}(f)$ 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 $\mathbb{V}_\mathcal{W}$. 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* $\mathbb{m}$, 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 $\rho \in [\![\,\tau\,]\!]_\mathbb{m}$ with $[\![\,\cdot\,]\!]_\mathbb{m}$ defined in §4.2. Intuitively, a well-formed region tag is a tree whose root (computed by function typ) is a type

Table 3. Memory model for TypedC.

Values	$v \in \mathbb{V}$	Addresses	$a \in \mathbb{A}$, $\mathbb{A} \subset (\mathbb{V}_{\mathcal{W}} \setminus \{0\}) \subset \mathbb{V}$
Allocation maps	$\mathbb{M} \ni \mathbb{m} : \mathbb{A} \rightharpoonup \mathbb{R}$	Region tags	$\mathbb{R} \ni \rho ::= \texttt{byte} \mid \tau/\vec{\rho}$ for $\tau \not\equiv \texttt{byte}$, τ closed
Byte allocation maps	$\overline{\mathbb{M}} \ni \overline{\mathbb{m}} : \mathbb{A} \rightharpoonup \overline{\mathbb{R}}$	Byte tags	$\overline{\mathbb{R}} \ni (\rho, k)$ s.t. $\rho \in \mathbb{R}, 0 \leq k < \text{size}(\rho)$
Type-offsets	$\mathbb{P} \triangleq \mathbb{T} \times \mathbb{N}$	Path in region tag	$\pi \in \mathbb{P}^+$

$$\text{size}(\rho) \triangleq \begin{cases} 1 & \text{if } \rho \equiv \texttt{byte} \\ \overline{\text{size}}(\vec{\rho}) & \text{if } \rho \equiv \tau/\vec{\rho} \end{cases} \qquad \overline{\text{size}}(\langle \rho_0 \rangle \cdot \vec{\rho}) \triangleq \text{size}(\rho_0) + \overline{\text{size}}(\vec{\rho})$$

$$\text{typ}(\rho) \triangleq \begin{cases} \texttt{byte} & \text{if } \rho \equiv \texttt{byte} \\ \tau & \text{if } \rho \equiv \tau/\vec{\rho}' \end{cases} \qquad \overline{\mathbb{m}}(a) \triangleq (\mathbb{m}(a_0), a - a_0) \text{ if } a_0 \in \text{dom}(\mathbb{m}) \text{ and } 0 \leq a - a_0 < \text{size}(\mathbb{m}(a_0))$$

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 $\tau_1 \times \ldots \times \tau_n$ (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 $\rho^1_{\texttt{node}}$ and $\rho^2_{\texttt{node}}$ 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 kth 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 $(\rho^1_{\texttt{node}}, 4)$ in Fig. 5 (for $\mathcal{W} = 4$). From a given allocation map $\mathbb{m}$, we define (in Table 3) the corresponding *byte allocation map* $\overline{\mathbb{m}}$ as a mapping from an address a to its byte tag built from the tag $\mathbb{m}(a_0)$ of the region starting at a_0 such that $a - a_0$ is positive and less than the size of the region (i.e., $\text{size}(\mathbb{m}(a_0))$). 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 $\mathbb{m}$ 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 sections[4] 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 $\mathbb{m}$ is well-formed iff $\forall \rho \in \text{img}(\mathbb{m})$ the root of ρ, i.e., $\text{typ}(\rho)$, is a type name η; ρ is well-formed, i.e., $\rho \in [\![\, \eta \,]\!]_{\mathbb{m}}$; and $\forall a_1, a_2 \in \text{dom}(\mathbb{m})$ the regions $\mathbb{m}(a_1)$ and $\mathbb{m}(a_2)$ are separated, i.e., $a_1 + \text{size}(\mathbb{m}(a_1)) \leq a_2 \lor a_2 + \text{size}(\mathbb{m}(a_2)) \leq a_1$.*

4.2 Denotations of Type Expressions

For a given allocation map $\mathbb{m}$, closed type expressions τ have a double interpretation in TypedC: as sets of values $(\!|\, \tau \,|\!)_{\mathbb{m}}$ and as sets of region tags $[\![\, \tau \,]\!]_{\mathbb{m}}$. 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 $(\!|\, \eta\star + e \,|\!)_{\mathbb{m}}$ contains the addresses a where a belongs to a region starting at address a_0, the region tag $\mathbb{m}(a_0)$ 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 $\mathbb{m}(\texttt{0x1000}) = \rho^2_{\texttt{node}}$, then $\texttt{0x1006} \in (\!|\, \texttt{int}\star{+}2 \,|\!)_{\mathbb{m}}$ because `int` appears at offset 4 in $\rho^2_{\texttt{node}}$. Formally, an address a belongs to $(\!|\, \eta\star + k \,|\!)_{\mathbb{m}}$ if the pair (η, k) appears in $\text{path}(\overline{\mathbb{m}}(a))$ defined as follows:

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

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

$\llparenthesis \cdot \rrparenthesis_{\mathbb{m}} : \mathbb{T} \to \mathcal{P}(\mathbb{V})$	$\llbracket \cdot \rrbracket_{\mathbb{m}} : \mathbb{T} \to \mathcal{P}(\mathbb{R})$
$\llparenthesis\, \mathtt{byte} \,\rrparenthesis_{\mathbb{m}} \triangleq \mathbb{V}_1$	$\llbracket\, \mathtt{byte} \,\rrbracket_{\mathbb{m}} \triangleq \{\mathtt{byte}\}$
$\llparenthesis\, \mathsf{n}(e_1, \ldots) \,\rrparenthesis_{\mathbb{m}} \triangleq \llparenthesis\, \Delta(\mathsf{n}(e_1, \ldots)) \,\rrparenthesis_{\mathbb{m}}$	$\llbracket\, \mathsf{n}(e_1, \ldots) \,\rrbracket_{\mathbb{m}} \triangleq \{\mathsf{n}(e_1, \ldots)/\rho \mid \rho \in \llbracket\, \Delta(\mathsf{n}(e_1, \ldots)) \,\rrbracket_{\mathbb{m}}\}$
$\llparenthesis\, \eta\star + e \,\rrparenthesis_{\mathbb{m}} \triangleq \{a \in \mathbb{A} \mid (\eta, \mathrm{eval}(e)) \in \mathrm{path}(\overline{\mathbb{m}}(a))\}$	$\llbracket\, \eta\star + e \,\rrbracket_{\mathbb{m}} \triangleq \{(\eta\star + e)/\mathtt{byte}[\mathcal{W}]\}$
$\llparenthesis\, \{x : \tau \mid e\} \,\rrparenthesis_{\mathbb{m}} \triangleq \{v \in \llparenthesis\, \tau \,\rrparenthesis_{\mathbb{m}} \mid \mathrm{eval}(e[x \leftarrow v]) \neq 0\}$	$\llbracket\, \{x : \tau \mid e\} \,\rrbracket_{\mathbb{m}} \triangleq \{(\{x : \tau \mid e\})/\rho \mid \rho \in \llbracket\, \tau \,\rrbracket_{\mathbb{m}}\}$
$\llparenthesis\, \tau_1 \times \tau_2 \,\rrparenthesis_{\mathbb{m}} \triangleq \{v_1 :: v_2 \mid v_i \in \llparenthesis\, \tau_i \,\rrparenthesis_{\mathbb{m}}\}$	$\llbracket\, \tau_1 \times \tau_2 \,\rrbracket_{\mathbb{m}} \triangleq \{(\tau_1 \times \tau_2)/\langle \rho_1 \rangle \cdot \langle \rho_2 \rangle] \mid \rho_i \in \llbracket\, \tau_i \,\rrbracket_{\mathbb{m}}\}$
$\llparenthesis\, \tau[e] \,\rrparenthesis_{\mathbb{m}} \triangleq \{v_0 :: \ldots :: v_{s-1} \mid$ $s = \mathrm{eval}(e),\ \forall i \in [0..s], v_i \in \llparenthesis\, \tau \,\rrparenthesis_{\mathbb{m}}\}$	$\llbracket\, \tau[e] \,\rrbracket_{\mathbb{m}} \triangleq \{(\tau[e])/\langle \rho_0 \rangle \cdot \ldots \cdot \langle \rho_{s-1} \rangle] \mid$ $s = \mathrm{eval}(e),\ \forall i \in [0..s].\ \rho_i \in \llbracket\, \tau \,\rrbracket_{\mathbb{m}}\}$
$\llparenthesis\, \tau_1 \cup \tau_2 \,\rrparenthesis_{\mathbb{m}} \triangleq \llparenthesis\, \tau_1 \,\rrparenthesis_{\mathbb{m}} \cup \llparenthesis\, \tau_2 \,\rrparenthesis_{\mathbb{m}}$	$\llbracket\, \tau_1 \cup \tau_2 \,\rrbracket_{\mathbb{m}} \triangleq \{(\tau_1 \cup \tau_2)/\rho \mid \rho \in \llbracket\, \tau_1 \,\rrbracket_{\mathbb{m}} \cup \llbracket\, \tau_2 \,\rrbracket_{\mathbb{m}}\}$
$\llparenthesis\, \exists \alpha : \tau_1 .\, \tau_2 \,\rrparenthesis_{\mathbb{m}} \triangleq \bigcup_{v_1 \in \llparenthesis\, \tau_1 \,\rrparenthesis_{\mathbb{m}}} \llparenthesis\, \tau_2[\alpha \leftarrow v_1] \,\rrparenthesis_{\mathbb{m}}$	$\llbracket\, \exists \alpha : \tau_1 .\, \tau_2 \,\rrbracket_{\mathbb{m}} \triangleq \{(\exists \alpha : \tau_1 .\, \tau_2)/\rho \mid$ $\exists v \in \llparenthesis\, \tau_1 \,\rrparenthesis_{\mathbb{m}}.\ \rho \in \llbracket\, \tau_2[\alpha \leftarrow v] \,\rrbracket_{\mathbb{m}}\}$
$\llparenthesis\, \tau_1 \mathbin{/\!\!\mid} \tau_2 \,\rrparenthesis_{\mathbb{m}} \triangleq \llparenthesis\, \tau_1 \,\rrparenthesis_{\mathbb{m}} \cap \llparenthesis\, \tau_2 \,\rrparenthesis_{\mathbb{m}}$	$\llbracket\, \tau_1 \mathbin{/\!\!\mid} \tau_2 \,\rrbracket_{\mathbb{m}} \triangleq \{(\tau_1 \mathbin{/\!\!\mid} \tau_2)/\rho_1 \mid \rho_1 \in \llbracket\, \tau_1 \,\rrbracket_{\mathbb{m}} \land \tau_2 \in \rho_1\}$

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

$$\begin{aligned} \mathrm{path}(\mathtt{byte}, 0) &\triangleq \langle(\mathtt{byte}, 0)\rangle \\ \mathrm{path}(\tau/(\vec{\rho}_1 \cdot \langle\rho\rangle \cdot \vec{\rho}_2), k) &\triangleq \langle(\tau, k)\rangle \cdot \mathrm{path}(\rho, k - \overline{\mathrm{size}}(\vec{\rho}_1)) \quad \text{if } \overline{\mathrm{size}}(\vec{\rho}_1) \leq k < \overline{\mathrm{size}}(\vec{\rho}_1 \cdot \langle\rho\rangle) \end{aligned}$$

The top-right part of Fig. 5 gives two paths for the byte tags $(\rho^1_{\mathtt{node}}, 10)$ and $(\rho^2_{\mathtt{node}}, 10)$ respectively. The above definition and the one of $\llparenthesis\, \eta\star{+}k \,\rrparenthesis_{\mathbb{m}}$ (Table 4) implies that this last set contains all addresses 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 start of this path). If we continue our previous example where $\overline{\mathbb{m}}(\mathtt{0x1010}) = (\rho^2_{\mathtt{node}}, 10)$, we can see that the address 0x1010 belongs to all of $\llparenthesis\, \mathtt{node}\star + 10 \,\rrparenthesis_{\mathbb{m}}$, $\llparenthesis\, \mathtt{inode}\star + 10 \,\rrparenthesis_{\mathbb{m}}$, and $\llparenthesis\, \mathtt{byte}\star + 0 \,\rrparenthesis_{\mathbb{m}}$.

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

The values of a type $\llparenthesis\, \tau \,\rrparenthesis_{\mathbb{m}}$ are not necessarily bit-vectors of the same size. For instance, consider an array type where elements have a size fixed by an existentially quantified variable like in the 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 $\mathbb{m}$ well-formed, $\forall \rho \in \llbracket\, \tau \,\rrbracket_{\mathbb{m}} : \mathrm{size}(\rho) = s$. If τ has a constant size, we denote it by $\mathrm{size}(\tau)$.

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.

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

$$\textsc{toDef}\quad \frac{\eta \neq \texttt{byte}}{(\Delta(\eta), k) \prec_{\mathbb{P}_{\mathbb{m}}} (\eta, k)} \qquad \textsc{toPtr}\quad \frac{}{(\texttt{byte}^{\mathcal{W}}, k) \prec_{\mathbb{P}_{\mathbb{m}}} (\eta\star + e, k)} \qquad \textsc{toRef}\quad \frac{}{(\tau, k) \prec_{\mathbb{P}_{\mathbb{m}}} (\{\texttt{self} : \tau \mid e\}, k)}$$

$$\textsc{toUnion1}\quad \frac{}{(\tau_1, k) \prec_{\mathbb{P}_{\mathbb{m}}} (\tau_1 \cup \tau_2, k)} \qquad \textsc{toUnion2}\quad \frac{}{(\tau_2, k) \prec_{\mathbb{P}_{\mathbb{m}}} (\tau_1 \cup \tau_2, k)} \qquad \textsc{toInter}\quad \frac{}{(\tau_1, k) \prec_{\mathbb{P}_{\mathbb{m}}} (\tau_1 \wedge \tau_2, k)}$$

$$\textsc{toProd1}\quad \frac{\rho_1 \in [\![\, \tau_1 \,]\!]_{\mathbb{m}} \qquad \mathrm{size}(\rho_1) > k}{(\tau_1, k) \prec_{\mathbb{P}_{\mathbb{m}}} (\tau_1 \times \tau_2, k)} \qquad \textsc{toProd2}\quad \frac{\rho_1 \in [\![\, \tau_1 \,]\!]_{\mathbb{m}} \qquad s = \mathrm{size}(\rho_1) \leq k}{(\tau_2, k - s) \prec_{\mathbb{P}_{\mathbb{m}}} (\tau_1 \times \tau_2, k)}$$

$$\textsc{toArr}\quad \frac{\vec{\rho} \cdot \langle \rho' \rangle \in [\![\, \tau \,]\!]^{+}_{\mathbb{m}} \qquad \overline{\mathrm{size}}(\vec{\rho}) \leq k < \overline{\mathrm{size}}(\vec{\rho} \cdot \langle \rho' \rangle)}{(\tau, k - \overline{\mathrm{size}}(\vec{\rho})) \prec_{\mathbb{P}_{\mathbb{m}}} (\tau[e], k)} \qquad \textsc{toEx}\quad \frac{v \in (\!|\, \tau_1 \,|\!)_{\mathbb{m}}}{(\tau_2[\alpha \leftarrow v], k) \prec_{\mathbb{P}_{\mathbb{m}}} (\exists \alpha : \tau_1 . \tau_2, k)}$$

$$\textsc{toProd1Cst}\quad \frac{\mathrm{size}(\tau_1) > k}{(\tau_1, k) \prec_{\mathbb{P}_{\mathbb{m}}} (\tau_1 \times \tau_2, k)} \qquad \textsc{toProd2Cst}\quad \frac{s = \mathrm{size}(\tau_1) \leq k}{(\tau_2, k - s) \prec_{\mathbb{P}_{\mathbb{m}}} (\tau_1 \times \tau_2, k)} \qquad \textsc{toArrCst}\quad \frac{s = \mathrm{size}(\tau)}{(\tau, k\%s) \prec_{\mathbb{P}_{\mathbb{m}}} (\tau[e], k)}$$

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 $\mathbb{m} \in \mathbb{M}$, we saw that the denotation of pointer types $(\!|\, \eta\star + e \,|\!)_{\mathbb{m}}$ provides a meaning to the pair of type name η and constant offset $\mathrm{eval}(e)$. 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 $\gamma_{\mathbb{P},\mathbb{m}}(\tau, k) \triangleq \{a \in \mathbb{A} \mid (\tau, k) \in \mathrm{path}(\overline{\mathbb{m}}(a))\}$. To obtain a sound abstraction, $\gamma_{\mathbb{P},\mathbb{m}}$ has to be monotonic [19]; this allows us to derive the *lattice of type-offset*, detailed later in this section. Observe that a concrete address a goes through 3 successive levels of abstractions: (i) the byte tag $(\rho, k) = \overline{\mathbb{m}}(a)$, (ii) the path $\pi = \mathrm{path}(\rho, k)$, 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 $\mathbb{m} \in \mathbb{M}$, we define in Table 5 a strict order relation, the *type-offset derivation* relation $\prec_{\mathbb{P}_{\mathbb{m}}} \subseteq \mathbb{P} \times \mathbb{P}$. The graph $(\mathbb{P}, \prec_{\mathbb{P}_{\mathbb{m}}})$ is called a *DAG of type-offsets*. Such a DAG is a sound abstraction of all the possible sequences in $\mathrm{path}(\rho, k)$ for all well-formed $\rho \in \mathbb{R}$ and all k s.t. $0 \leq k < \mathrm{size}(\rho)$.

Intuitively, the relation $\prec_{\mathbb{P}_{\mathbb{m}}}$ follows the derivation relation $\prec_{\Delta}$ on type expressions (see Fig. 2), adding the offset k. The relation between offsets in $\prec_{\mathbb{P}_{\mathbb{m}}}$ 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 [70, App. A] that $\prec_{\mathbb{P}_{\mathbb{m}}}$ is well-founded when $\prec_{\Delta}$ is well-founded. Using $\prec_{\mathbb{P}_{\mathbb{m}}}$, we define in Table 6 the set of paths in the DAG of type-offsets, $\mathbb{P}^{+}_{\mathbb{m}}$, where each path ends in $(\texttt{byte}, 0)$ (as paths of well-formed region tags end in byte). We also prove in [70, App. A]

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

$$\mathbb{P}^+_{\mathbb{m}} \triangleq \{\pi \in \mathbb{P}^+ \mid \forall i.\ \pi[i+1] \prec_{\mathbb{P}_{\mathbb{m}}} \pi[i] \wedge \text{last}(\pi) = (\texttt{byte}, 0)\}$$

$$\gamma_{\mathbb{P}^+,\mathbb{m}}(\tau, k) \triangleq \{\pi \in \mathbb{P}^+_{\mathbb{m}} \mid (\tau, k) \in \pi\}$$

$$\mathbb{P}_{\mathbb{m}} \triangleq \{(\tau, k) \in \mathbb{P} \mid \exists \pi \in \mathbb{P}^+_{\mathbb{m}}.\ (\tau, k) \in \pi\}$$

$$\gamma_{\mathbb{P},\mathbb{m}}(\tau, k) \triangleq \{a \in \mathbb{A} \mid \overline{\mathbb{m}}(a) \in \gamma_{\mathbb{P}^+,\mathbb{m}}(\tau, k)\}$$

$$\gamma_{\mathbb{P}}(\tau, k) \triangleq \{a \in \mathbb{A} \mid \exists \mathbb{m}.\ a \in \gamma_{\mathbb{P},\mathbb{m}}(\tau, k)\}$$

that $\mathbb{P}^+_{\mathbb{m}}$ over-approximates the set $\{\overline{\mathbb{m}}(a) \mid a \in \text{dom}(\mathbb{m})\}$ of paths in $\mathbb{m}$. Finally, we define $\mathbb{P}_{\mathbb{m}}$, which restricts $\mathbb{P}$ to the type-offsets found in the DAG of type-offsets for $\mathbb{m}$.

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

In Table 6, we decompose the definition of $\gamma_{\mathbb{P},\mathbb{m}}$ using the concretization $\gamma_{\mathbb{P}^+,\mathbb{m}}$ of a type-offset into the set of paths that traverse it. This allows us to prove (see [70, App. A]) the following fundamental theorems for physical subtyping [11] and type-based alias analysis [24] in our type system.

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

(1) *If* $(\tau_1, k_1) \sqsubseteq_{\mathbb{P}_{\mathbb{m}}} (\tau_2, k_2)$ *then* $\gamma_{\mathbb{P},\mathbb{m}}(\tau_1, k_1) \subseteq \gamma_{\mathbb{P},\mathbb{m}}(\tau_2, k_2)$, *i.e., a dominating type-offset includes the addresses of the dominated type-offset.*

(2) *If* $\gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_1, k_1) \cap \gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_2, k_2) = \emptyset$ *then* $\gamma_{\mathbb{P},\mathbb{m}}(\tau_1, k_1) \cap \gamma_{\mathbb{P},\mathbb{m}}(\tau_2, k_2) = \emptyset$, *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 $\mathbb{m}$ in Fig. 5, we have $(\texttt{int}, 0) \prec_{\mathbb{P}_{\mathbb{m}}} (\texttt{inode}, 4)$ and $(\texttt{int}, 0) \prec_{\mathbb{P}_{\mathbb{m}}} (\texttt{node}, 4)$. Because $(\texttt{int}, 0)$ is the least common predecessor of each in the type-offset DAG, it is the immediate dominator, and thus the result of $(\texttt{inode}, 4) \sqcup_{\mathbb{P}_{\mathbb{m}}} (\texttt{node}, 4)$. Furthermore, pointers typed by $(\texttt{inode}, 4)$ and $(\texttt{node}, 8)$ do not alias as there is no path shared between these type-offsets.

Abstracting over allocation maps. Both relations defined above, $\prec_{\mathbb{P}_{\mathbb{m}}}$ and $\sqsubseteq_{\mathbb{P}_{\mathbb{m}}}$, depend on an allocation map. However, this allocation map is not known during the analysis. Therefore, we abstract it by defining $\prec_{\mathbb{P}} \triangleq \bigcup_{\mathbb{m} \in \mathbb{M}} \prec_{\mathbb{P}_{\mathbb{m}}}$. The direct definition of $\prec_{\mathbb{P}}$ is simple under the constant prefix hypothesis: just replace $\prec_{\mathbb{P}_{\mathbb{m}}}$ by $\prec_{\mathbb{P}}$ in Table 5, and rewrite the premise of the rule TOEX as "$\exists \mathbb{m} \in \mathbb{M}$ well-formed, $v \in (\!| \tau_1 |\!)_{\mathbb{m}}$" to be independent of a fixed $\mathbb{m}$. 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 $\langle \mathbb{P}_{\mathbb{m}}, \sqsubseteq_{\mathbb{P}_{\mathbb{m}}}, \sqcup_{\mathbb{P}_{\mathbb{m}}} \rangle$.

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 [70, App. B]) 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 writes to memory (stores) maintain the invariant.

Concrete states. In an untyped execution semantic, program states s are pairs $(\sigma, \text{h}) \in \mathbb{S}$ of a store $\mathbb{\Sigma} \ni \sigma : \mathbb{X} \rightarrow \mathbb{V}$ mapping program variables (including registers) in $\mathbb{X}$ to values, and a heap $\text{h} : \mathbb{A} \rightharpoonup \mathbb{V}_1$ 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 $\mathbb{m}$. Thus, for a given $\mathbb{m}$, the set of values that can be stored at addresses tagged by $\rho \in \mathbb{R}$ is defined as follows:

$$\llparenthesis \texttt{byte} \rrparenthesis_{\mathbb{m}} \triangleq \mathbb{V}_1 \qquad \llparenthesis \tau/\vec{\rho}\, \rrparenthesis_{\mathbb{m}} \triangleq \llparenthesis \tau \rrparenthesis_{\mathbb{m}} \cap \{v_0 :: \ldots :: v_{n-1} \mid v_i \in \llparenthesis \vec{\rho}[i] \rrparenthesis_{\mathbb{m}}, i \in [0..n]\} \tag{1}$$

A *well-typed heap* is a pair $(\mathbb{m}, h)$ where $\mathbb{m}$ is well-formed, $\mathrm{dom}(h) = \mathrm{dom}(\overline{\mathbb{m}})$ and for all $a \in \mathrm{dom}(\mathbb{m})$, $h(a) :: \ldots :: h(a + \mathrm{size}(\mathbb{m}(a)) - 1) \in \llparenthesis \mathbb{m}(a) \rrparenthesis_{\mathbb{m}}$. 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 $[a..a + \ell] \in \mathrm{dom}(h)$. We denote by $\mathrm{h}[a..a+\ell]$ the *load from heap* operation which returns the bit vector of size ℓ obtained by concatenation of values $h(a) :: h(a + 1) :: \ldots :: h(a + \ell - 1)$. The *store to heap* operation at address a on ℓ consecutive values with the new value $v \in \mathbb{V}_\ell$ is denoted by $\mathrm{h}[a..a + \ell \longleftarrow v]$.

Proposition 2 (Typed Load). *Let $(\mathbb{m}, h)$ be a well-typed heap and a be an address such that $[a..a + \ell] \subseteq \mathrm{dom}(h)$ for some ℓ and $(\rho, k) = \overline{\mathbb{m}}(a)$. Then $\mathrm{h}[a..a + \ell] \in \{\, v[k..k + \ell] \mid v \in \llparenthesis \rho \rrparenthesis_{\mathbb{m}} \,\}$.*

The above proposition enables inferring properties about the contents of the heap. Suppose a C function which receives a pointer `p` of type `node`★ (like `extend` in Fig. 5). If we load 4 bytes from `p`+8, we know that we will receive a value v in $\llparenthesis \texttt{node}\star \rrparenthesis_{\mathbb{m}}$ or in $\llparenthesis \texttt{nullptr} \rrparenthesis_{\mathbb{m}}$ (depending on the initial value of $(\mathbb{m}, h)$). Furthermore, if the test v equals 0 succeeds, because $0 \notin \llparenthesis \texttt{node}\star \rrparenthesis_{\mathbb{m}}$ we deduce that the region corresponding to `p` cannot have $\rho^2_{\texttt{node}}$ as a region tag, and that a value obtained by another load at `p`+12 would also be in $\llparenthesis \texttt{nullptr} \rrparenthesis_{\mathbb{m}}$ (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 $(\mathbb{m}, h)$ be a well-typed heap, $a \in \mathrm{dom}(h)$ and $v \in \mathbb{V}_\ell$. The store $\mathrm{h}[a..a + \ell \longleftarrow v]$ is safe if there exists $\mathbb{m}'$ such that:

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

Intuitively, condition (1) above corresponds to finding an allocation map such that the heap is well-typed after the store. The resulting allocation map may differ at addresses outside of the $[a..a + \ell]$ range. For instance, 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 $(\mathbb{m}, h)$ be a well-typed heap, $a \in \mathrm{dom}(h)$ an allocated address such that $(\rho, k) \in \overline{\mathbb{m}}(a)$ and $s = \mathrm{size}(\rho)$. Let $v \in \mathbb{V}_\ell$ be a value. Then $h[a..a+\ell \longleftarrow v]$ is a safe store iff $h[a - k..a - k + s \longleftarrow h[a - k..a] :: v :: h[a + \ell..a - k + s]]$ is a safe store.*

Condition (2) of Def. 6.1 corresponds to type preservation. Following the definitions in Table 4, it is easy to prove that condition (2) amounts to a monotony condition on the pointer types, i.e., $\forall \eta, k : \llparenthesis \eta\star + k \rrparenthesis_{\mathbb{m}} \subseteq \llparenthesis \eta\star + k \rrparenthesis_{\mathbb{m}'}$. This is easily proved when $\mathbb{m}' = \mathbb{m}$, which corresponds to a weak update. Arbitrary changes to $\mathbb{m}$, as in a strong update (that we do not allow) would break this condition: given $a \in \llparenthesis \eta\star + k \rrparenthesis_{\mathbb{m}}$, we can change $\mathbb{m}(a)$ so that $a \notin \llparenthesis \eta\star + k \rrparenthesis_{\mathbb{m}'}$ after the update. Consider a pointer `p` of type `inode`★ in Fig. 5. The judgment `p`:`inode`★ would no longer be true if we wrote 0 at `p`+8 and `p`+12. Thus, because pointers to interior nodes may exist, we cannot transform them into leaf nodes, as this would require a strong update.

Mild updates are those that change $\mathbb{m}$, the invariant on the contents of the heap, without breaking existing typing judgments. They are possible mainly because pointers are to type names, and not to arbitrary types. For instance, suppose that the function `extend` in Fig. 5 modifies a leaf node to

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

$\alpha \in \mathbb{\alpha}$ (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)	
$\nu \in \mathbb{\nu} \triangleq \mathbb{\alpha} \to \mathbb{V}$ (valuation)	$\hat{\mathbb{T}} \ni \hat{\tau} ::= \eta \mid \hat{\tau}\star + \hat{e} \mid \{\mathtt{self} : \hat{\tau} \mid \hat{e}\} \mid \hat{\tau}_1 \times \hat{\tau}_2 \mid \ldots$ (symbolic types)	
$\nu^\sharp \in \mathbb{\nu}^\sharp$ (numerical domain)	$\Gamma^\sharp \in \mathbb{\Gamma}^\sharp \triangleq \hat{\mathbb{E}} \rightharpoonup (\hat{\mathbb{T}} \times \hat{\mathbb{E}})$ (abstract type environment)	
$x \in \mathbb{X}$ (prog. variables)	$\sigma^\sharp \in \Sigma^\sharp \triangleq \mathbb{X} \to \hat{\mathbb{E}}$ (abstract store)	$s^\sharp \in \mathbb{S}^\sharp ::= \Sigma^\sharp \times \mathbb{\Gamma}^\sharp \times \mathbb{\nu}^\sharp$ (abstract state)
$\gamma_{\hat{\mathbb{E}}} : \hat{\mathbb{E}} \to (\mathbb{\nu} \to \mathbb{V})$	$\gamma_{\hat{\mathbb{E}}}(\hat{e}) \triangleq \lambda \nu.\ \mathrm{eval}(\mathrm{subst}(\hat{e}, \nu))$	
$\gamma_{\hat{\mathbb{T}}} : \hat{\mathbb{T}} \to (\mathbb{M} \times \mathbb{\nu}) \to \mathcal{P}(\mathbb{V})$	$\gamma_{\hat{\mathbb{T}}}(\hat{\tau}) \triangleq \lambda(\mathbb{m}, \nu).\ \llparenthesis\, \mathrm{subst}(\hat{\tau}, \nu)\, \rrparenthesis_{\mathbb{m}}$ with $\llparenthesis\, \tau\star + e\, \rrparenthesis_{\mathbb{m}} \triangleq \{a \in \mathbb{A} \mid (\tau, \mathrm{eval}(e)) \in \mathrm{path}(\overline{\mathbb{m}}(a))\}$	
$\gamma_{\mathbb{\Gamma}^\sharp} : \mathbb{\Gamma}^\sharp \to \mathbb{M} \to \mathcal{P}(\mathbb{\nu})$	$\gamma_{\mathbb{\Gamma}^\sharp}(\Gamma^\sharp) \triangleq \lambda \mathbb{m}. \bigcap_{(\hat{e}_1 \mapsto (\hat{\tau}, \hat{e}_2)) \in \Gamma^\sharp} \{\nu \in \mathbb{\nu} \mid \gamma_{\hat{\mathbb{E}}}(\hat{e}_1)(\nu) \in \gamma_{\hat{\mathbb{T}}}(\hat{\tau}\star + \hat{e}_2)(\mathbb{m}, \nu)\}$	
$\gamma_{\mathbb{\nu}^\sharp} : \mathbb{\nu}^\sharp \to \mathcal{P}(\mathbb{\nu})$	(given by the numerical domain)	
$\gamma_{\Sigma^\sharp} : \Sigma^\sharp \to (\mathbb{\nu} \to \Sigma)$	$\gamma_{\Sigma^\sharp}(\sigma^\sharp) \triangleq \lambda \nu.\ \lambda x.\ \gamma_{\hat{\mathbb{E}}}(\sigma^\sharp(x))(\nu)$	
$\gamma_{\mathbb{M}^\sharp} : \Sigma^\sharp \to (\mathbb{M} \times \mathbb{\nu}) \to \mathcal{P}(\mathbb{S})$	$\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp) \triangleq \lambda(\mathbb{m}, \nu).\ \{(\sigma, h) \mid \sigma \in \gamma_{\Sigma^\sharp}(\sigma^\sharp)(\nu) \wedge (\mathbb{m}, h) \text{ well-typed}\}$	
$\gamma_{\mathbb{S}^\sharp} : \mathbb{S}^\sharp \to \mathcal{P}(\mathbb{S})$	$\gamma_{\mathbb{S}^\sharp}(\sigma^\sharp, \Gamma^\sharp, \nu^\sharp) \triangleq \bigcup_{\mathbb{m} \in \mathbb{M}, \nu \in \gamma_{\mathbb{\Gamma}^\sharp}(\Gamma^\sharp)(\mathbb{m}) \cap \gamma_{\mathbb{\nu}^\sharp}(\nu^\sharp)} \gamma_{\mathbb{M}^\sharp}(\sigma^\sharp)(\mathbb{m}, \nu)$	

turn it into an interior node by changing the two last fields from null to non-null pointers. As there is no name for the leaf node type, no "pointer to leaf node" type exists. Then, the fact that `extend` decreases the number of leaf nodes in $\mathbb{m}'$ is not a problem. We also have $\llparenthesis\, \mathtt{node}\star\, \rrparenthesis_{\mathbb{m}} = \llparenthesis\, \mathtt{node}\star\, \rrparenthesis_{\mathbb{m}'}$ as the type `node`★ cannot distinguish a pointer to an interior node from a pointer to a leaf node. Finally, $\llparenthesis\, \mathtt{inode}\star\, \rrparenthesis_{\mathbb{m}'}$ increases, which is allowed. Thus, `extend` is well-typed.

To verify the monotony, we just verify that the paths in the new allocation map $\mathbb{m}'$ contain all the type-offset pairs (η, k), for which the type is a type name, that were in the previous allocation map $\mathbb{m}$ (i.e., all the paths must go through the same "circled nodes" in Fig. 5).

Theorem 6.3 (Name matching implies monotony). *Let* $\mathbb{m}, \mathbb{m}' \in \mathbb{A} \rightharpoonup \mathbb{R}$.

If $\quad \forall a \in \mathrm{dom}(\mathbb{m}) : \quad \{(\eta, k) \in \mathrm{path}(\overline{\mathbb{m}}(a))\} \subseteq \{(\eta, k) \in \mathrm{path}(\overline{\mathbb{m}}'(a))\},$

Then $\quad \forall \tau \in \mathbb{T} : \quad \llparenthesis\, \tau\, \rrparenthesis_{\mathbb{m}} \subseteq \llparenthesis\, \tau\, \rrparenthesis_{\mathbb{m}'}$ *and* $[\![\, \tau\,]\!]_{\mathbb{m}} \subseteq [\![\, \tau\,]\!]_{\mathbb{m}'}$.

For instance, a call to function `to_leaf` in Fig. 4 transforms an interior node at address `p` whose region tag is $\rho^2_{\mathtt{node}}$ to a leaf node. To make the resulting heap well-typed, the new region tag of `p` shall be $\rho^1_{\mathtt{node}}$. But $\mathrm{path}(\rho^1_{\mathtt{node}}, 10)$ in Fig. 5 does not contain the type-offset $(\mathtt{inode}, 10)$, while $\mathrm{path}(\rho^2_{\mathtt{node}}, 10)$ does, so we reject this change. The opposite change, like in the `extend` function, makes every path go through more circled nodes, so it is well-typed.

7 Type-Checking by Abstract Interpretation

Our analysis is defined as an abstract interpretation [18]. This section shortly presents the definition of our abstract domains and their meaning using concretization functions γ, some abstract transformers and their soundness. The detailed definition of abstract transformers and their properties is given in [70, App. C].

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 $\mathbb{\alpha}$ of *symbolic variables*. A *valuation* $\nu \in \mathbb{\nu}$ is a mapping from symbolic variables to values. Given a valuation ν, we represent by $\mathrm{subst}(\hat{e}, \nu)$ 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 $(\!| \tau\star + e |\!)_{\mathbb{m}}$ that extends Table 4. The concretization function $\gamma_{\hat{\mathbb{T}}}$ first concretizes a symbolic type $\hat{\tau}$ to a concrete type $\tau = \mathrm{subst}(\hat{\tau}, \nu)$ given a valuation ν; then, $\gamma_{\hat{\mathbb{T}}}$ uses the set of values $(\!| \tau |\!)_{\mathbb{m}}$ for a given allocation map $\mathbb{m}$. Thus, $\gamma_{\hat{\mathbb{T}}}$ is a function which inputs both a valuation ν and an allocation map $\mathbb{m}$.

Abstract type environments $\Gamma^{\sharp}$ 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 $\Gamma^{\sharp}$ 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 $s^{\sharp} \vDash \hat{e} : \hat{\tau}\star + \hat{e}'$ (where $\Gamma^{\sharp}$ is a component of $s^{\sharp}$). Noteworthy, $\Gamma^{\sharp}$ 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 $\nu^{\sharp}$ 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 $\Gamma^{\sharp}$ can be interpreted (using $\gamma_{\Gamma^{\sharp}}$) as a set of constraints on the valuations $\mathbb{\nu}$. For instance, if we derive using $\Gamma^{\sharp}$ 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 $\gamma_{\Gamma^{\sharp}}$ is an allocation map $\mathbb{m}$ used to interpret symbolic types as sets of values.

As mentioned above, $\Gamma^{\sharp}$ 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 $s^{\sharp} \vDash \hat{e} : \hat{\tau}\star + \hat{e}_2$ where $\hat{\tau}$ is not a type name (for instance in Fig. 5, if we have $s^{\sharp} \vDash \hat{e} : \texttt{node}\star$ and the value loaded at $\hat{e} + 8$ is null, we can infer that $\hat{e}$ points to a leaf node, i.e. $s^{\sharp} \vDash \hat{e} : \big(\texttt{inode}? \times \texttt{int} \times \texttt{nullptr} \times \texttt{nullptr}\big)\star$). 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 $s^{\sharp} \vDash \hat{e} : \eta\star + \hat{e}'$. Thus, in the formalization of abstract transformers (see [70, App. C]), we limit $\Gamma^{\sharp}$ to contain mappings to pairs of the form $(\eta, \hat{e})$. This means that judgments of the form $s^{\sharp} \vDash \hat{e} : \hat{\tau}\star + \hat{e}_2$ can be made but cannot be saved in $\Gamma^{\sharp}$, which makes sound the abstract semantics of store on heap operations.

The *numerical domain* $\mathbb{\nu}^{\sharp}$ complements the set of constraints of $\Gamma^{\sharp}$. Elements of $\mathbb{\nu}^{\sharp}$ are numerical constraints over symbolic terms (such as $3 \leq \alpha + 1 \leq 7$), and are interpreted by $\gamma_{\mathbb{\nu}^{\sharp}}$ as a set of valuations that match all these constraints. All the classical numerical domains (e.g., intervals [18], congruences [31], octagons [52],...) can be used.

The memory is abstracted using *abstract stores* $\sigma^{\sharp}$, which map program variables $\mathbb{X}$ to symbolic expressions $\hat{\mathbb{E}}$, an abstract counterpart to the concrete stores. Abstract stores are concretized by $\gamma_{\Sigma^{\sharp}}$ to stores using a valuation of the symbolic expressions. There is no abstract counterpart to the concrete heap. Indeed, one of the goals of our type system is to define invariants on the memory that allow the program analysis without a representation of the heap (unlike, for instance shape analysis), in order to obtain fast analysis operations. However, given an abstract store $\sigma^{\sharp}$, an allocation map $\mathbb{m}$ and a valuation ν, we can define the set of all the possible corresponding concrete states (σ, h)

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

$$\textsc{Const}: \{s_0^\sharp\}\ k \Downarrow k\ \{s_0^\sharp\} \qquad \textsc{Var}: \{s_0^\sharp\}\ x \Downarrow s_0^\sharp.\sigma^\sharp[x]\ \{s_0^\sharp\} \qquad \textsc{Load}: \frac{\{s_0^\sharp\}\ E \Downarrow \hat{e}_1\ \{s_1^\sharp\} \qquad \{s_1^\sharp\}\ *_\ell\ \hat{e}_1 \Downarrow \hat{e}_2\ \{s_2^\sharp\}}{\{s_0^\sharp\}\ *_\ell\ E \Downarrow \hat{e}_2\ \{s_2^\sharp\}}$$

$$\textsc{Binop}\ \diamond: \frac{\{s_0^\sharp\}\ E_1 \Downarrow \hat{e}_1\ \{s_1^\sharp\} \qquad \{s_1^\sharp\}\ E_2 \Downarrow \hat{e}_2\ \{s_2^\sharp\} \qquad s_2^\sharp \vDash \hat{e}_2 \neq 0\ \ (\text{if } \diamond = /)}{\{s_0^\sharp\}\ E_1 \diamond E_2 \Downarrow \hat{e}_1 \diamond \hat{e}_2\ \{s_2^\sharp\}} \qquad \textsc{PtrAdd}: \frac{s^\sharp \vDash \hat{e} : (\hat{\tau}\star + \hat{e}_1)}{s^\sharp \vDash (\hat{e} + \hat{e}_2) : \hat{\tau}\star + (\hat{e}_1 + \hat{e}_2)}$$

$$\textsc{LoadSimple}: \frac{s_0^\sharp \vDash \hat{e}_1 : (\hat{\tau}\star + \hat{e}_2) \qquad s_0^\sharp \vDash \mathrm{size}(\hat{\tau}) = \ell \qquad s_0^\sharp \vDash \hat{e}_2 = 0 \qquad \alpha \text{ fresh} \qquad s_0^\sharp \wedge \alpha : \hat{\tau} \Rightarrow s_1^\sharp}{\{s_0^\sharp\}\ *_\ell \hat{e}_1 \Downarrow \alpha\ \{s_1^\sharp\}}$$

$$\textsc{LoadLarger}: \frac{s_0^\sharp \vDash \hat{e}_1 : (\hat{\tau}\star + \hat{e}_2) \qquad s_0^\sharp \vDash \hat{e}_2 = k \qquad s_0^\sharp \vDash \mathrm{size}(\hat{\tau}) = \ell_2 \qquad \ell_2 \geq k + \ell_1 \qquad \{s_0^\sharp\}\ *_{\ell_2}(\hat{e}_1 - k) \Downarrow \hat{e}_3\ \{s_1^\sharp\}}{\{s_0^\sharp\}\ *_{\ell_1} \hat{e}_1 \Downarrow \hat{e}_3[k..k + \ell_1]\ \{s_1^\sharp\}}$$

using the function $\gamma_{\mathbb{M}^\sharp}$ (where $\mathbb{M}^\sharp$ stands for memory). Noteworthy, the only constraint on the heap is that $(\mathbb{m}, h)$ is well-typed, which implies that $\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp)$ will be empty if $\mathbb{m}$ is not well-formed.

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

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

7.2 Flow-Sensitive Analysis

Our analysis is derived from the definition of our abstract domain and its concretisation, as usual in abstract interpretation [17, §45]. The basic transformers required by the abstract domain (reading from memory, storing in memory, computing expressions, testing conditions) involve additional operations to interchange information between the different parts of the analysis. We present all these operations and their soundness theorem in [70, App. C]. As an illustration, we provide in Table 8 some of the rules necessary to handle the evaluation of expressions, including loading from memory, in a simple language whose semantics is close to that of machine code.

We adopt an axiomatic formalization of the abstract transformers, where different logical assertions correspond to different modules of the analyzer (in particular, abstract domains). For instance, the abstract evaluation of a program expression E into a symbolic expression $\hat{e}$ [42] starting from 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 by inference rules, so a proof tree may be used as a witness for type-safety [56, 59]. The actual implementation of the analysis (see §8) is a forward abstract interpretation derived from these 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. Noteworthy, the implementation does not need to perform any backtracking proof search.

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

When a new symbolic expression, like $\hat{e} + \hat{e}_2$, is created, new information is attached to it in the numerical or in the type domain. The rule PtrAdd illustrates the case of the type domain, which infers symbolic types for symbolic expressions, i.e., assertions $s^\sharp \vDash \hat{e} : \hat{\tau}$. When a program 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 $\{s_0^\sharp\}\ *_\ell\ \hat{e}_1 \Downarrow \hat{e}_2\ \{s_1^\sharp\}$, 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* $\{s_1^\sharp\}\ E \Downarrow \hat{e}\ \{s_2^\sharp\}$ *and* $s \in \gamma_{\mathbb{S}^\sharp}(s_1^\sharp)$ *and* $s \vdash E \Downarrow v$. *Let* $(\mathfrak{m}, \nu)$ *such that* $s \in \gamma_{\mathbb{M}^\sharp}(s_1^\sharp)(\mathfrak{m}, \nu)$. *Then* $s \in \gamma_{\mathbb{S}^\sharp}(s_2^\sharp)$ *and* $s \in \gamma_{\mathbb{M}^\sharp}(s_2^\sharp)(\mathfrak{m}, \nu)$ *and* $\gamma_{\hat{\mathbb{E}}}(\hat{e})(\nu) = v$.

Lemma 7.2 (Numerical queries). *If* $s^\sharp \vDash \hat{e}$, $s \in \gamma_{\mathbb{S}^\sharp}(s^\sharp)$ *and* $s \in \gamma_{\mathbb{M}^\sharp}(s^\sharp)(\mathfrak{m}, \nu)$, *then* $\gamma_{\hat{\mathbb{E}}}(\hat{e})(\nu) \neq 0$.

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, 48, 68]?

Implementation and experimental setup. We implemented our analysis as part of Codex, 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 $\sigma^\sharp$ are replaced with a domain more suitable to the languages we handle. For C, Codex relies on the platform Frama-C to parse C files into a CFG and replaces $\sigma^\sharp$ with a flow-sensitive representation of addressable C local and global variables (similar to [51]). For machine code, Codex relies on the platform Binsec to translate machine code instructions into a simpler intermediate language, and replaces $\sigma^\sharp$ with a flow-sensitive abstraction of the stack, registers and global variables. Codex also includes in $s^\sharp$ several abstract domains to reconstruct the control-flow graph [6, 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 [5] (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);

(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 [15, 48, 58, 60, 66]. Each case study is superscripted by `bin` or `C` when only one kind of code is analyzed.[5] 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.

```
typedef struct node {
  int value;
  struct node *left;
  struct node *right;
} node;
... // Other functions
node* RandTree(int, int, int, int);
int Bimerge(node*, int, int);
int Bisort(node*, int, int);
```

```
def node_pu(h) := union {
  (node(h)+ with h>0) case1;
  ((int with self==0) with h==0) case2;
};
def node(h) := struct {
  int value;
  node_pu(h-1) left;
  node_pu(h-1) right;
};
def node_pp := ∃h:(int with self>0).node(h)+
def node_pz := ∃h:(int with self>0).node(h)?
... // Other functions
node_pz RandTree(int, int, int, int);
int Bimerge(node_pp, int, int);
int Bisort(node_pp, int, int);
```

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

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

[5] Some 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**
				BS	DU	NLI	FAM	IP	P?	gen	man	gen	final	true	(s)
OS	Contiki	329	12	–	–	–	–	–	✓	19	14	16	2	0	1.33
	QDS$^{\text{bin}}$	401	3	–	✓	✓	–	–	✓	83	83	18	0	0	1.28
	RBTree Linux	1 111	2	–	–	–	–	✓	✓	29	17	6	2	0	0.46
Emacs	list$^{\text{bin}}$	464	8	✓	✓	–	–	–	✓		73	-	0	0	3.03
	string$^{\text{bin}}$	109	5	✓	✓	✓	–	–	✓			-	4	0	3.20
	buffer$^{\text{bin}}$	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

running our analysis with this new specification file, 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 [12, 46, 47].

We also compare with CheckedC [28, 48], a state-of-the-art tool to make C code spatially memory-safe. *Qualitatively*, one using CheckedC has to perform three kinds of *code modifications*: (1) modification of the C code to handle limitations in the type system [48] like absence of unions, 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 the annotation generator 3C [48]. 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 [48] the number of changes made by CheckedC 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 (i.e., column final contains less alarms than column gen). *Final* alarms are those we cannot prove, even with our most precise specification, due (mostly) to analysis imprecision or (sometimes) to the lack of expressivity in our type system. These alarms could be eliminated if we modified the code to insert run-time checks, like CheckedC does (for this reason CheckedC does not report alarms). These checks would degrade execution performance, but much less than when

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

Case studies	**#LoC**	**#Entry**	**Code patterns**						**CC+3C**		**Spec**		**#Alarms**			**Time**
(Olden)			BS	DU	NLI	FAM	IP	P?	man	gen	man	gen	gen	final	true	(s)
bh^C	2 107	30	–	✓	–	–	–	✓	181	48	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	–	–	–	–	–	✓9	83	30	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	9	2	32	6	0	0	3.86
voronoi^C	1 151	40	✓	–	–	–	–	✓	✗	✗	38	101	57	44	0	21.35

the checks are systematically inserted, as in CheckedC or other dynamic methods. *True* alarms in Shapes/Javl, Olden/bh and Olden/mst have been confirmed by manual inspection and correspond to null pointer dereferences.

Conclusions. 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 CheckedC (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 [70, 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 [8, 9] where the memory for different types is entirely separate, or abstract models like the CompCert memory model [44, 45] 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 [43, 49] 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 [76]. 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 [73]. Ahmed [2, §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, 74]) was one of the main reasons behind the rise of the syntactic method to prove type soundness [79]. For semantic proofs, a generic solution is step-indexed logical relations [2], 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. [10, 13, 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 [3, 55, 57, 80] 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 [15, 81] 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 the program. Some of these type systems support nominal nesting of structures, and Deputy also supports some variant types. It is also common to reason about structure nesting to derive aliasing information [24]. What we add in this area is our notions of region tags, byte tags, and the lattice of type offsets, that extends type-based reasoning about aliases to type systems allowing both concatenation and union of regions, and allows byte-level reasoning [51] about memory.

Run-time and hybrid type checking: Some approaches [25, 37] propose run-time type checking for low-level languages like C/C++ which also guarantees memory safety. These approaches correspond to the implementation of the concrete semantics (see §6) for a simplified version of our type system. Because dynamic type checking methods have a run-time access to the allocation map, they can deal with temporal memory safety and strong updates with a simpler type system that does not include refined, existential or parameterized types, and all symbolic variables in types are replaced by run-time constants. The drawback is a significant overhead at execution time and no proof that the program will not crash with a type error during execution.

Type checking using abstract interpretation: Flow-insensitive syntactic type checking suffers from inherent limitations [28] which are solved by annotating, porting the code, and inserting dynamic checks. To alleviate this problem, automated porting tools have been developed for CCured [60] and CheckedC [48, 68]. An alternative is to make the type checking algorithm more precise, which is necessary when we cannot modify the source code, like in machine code analysis.

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

The analysis of Nicole et al. [61] is the closest to ours, as it also combines abstract interpretation, physical and refinement types to prove spatial memory safety. We extend their work by also supporting unions (using union and existential types) and relations (using parameterized types) between regions. This makes the type system, semantic model, and aliasing rules significantly more complex. In particular, the main challenge when developing our type system and its semantic model came from the interaction between union and concatenation of regions. Viewing types as program abstractions has been studied theoretically in [16], and done in very different contexts [54].

Alias and memory analyses: There are many analyses that assume that a program is well-typed to derive aliasing information [24, 62]. Using abstract interpretation to combine analyses [19], we simultaneously prove that programs are well-typed and use this information to compute more precise alias information, which helps type checking the program. There are flow-sensitive memory analyses, e.g. [51], for low-level abstraction of memory regions, or shape analyses for low-level code [26, 39, 41]) that can compute expressive invariants about memory. A type-based analysis like ours, may be less precise but it is faster and more easily made modular, because the complex heap invariants are implicitly represented by the flow-insensitive type definitions.

10 Conclusion

We have presented a rich non-substructural type system which is *physical* (types represent a memory layout), *nominal* (pointers to different names do not alias just because they hold the same content), and *dependent* (it captures complex relations between values and memory layout, like array sizes). This type system is expressive enough to express many low-level programming patterns found in C and binary code. We have used this type system to implement a modular abstract interpreter [20] for low-level code, that performs an automated semantic type checking against user-provided prototypes. Thanks to abstract interpretation, we can perform type checking and inference in practice despite the undecidability of type-checking with dependent types. Our type-checking implies spatial memory safety and has promising results on challenging benchmarks, allowing us to verify code with a reduced effort.

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 [21] 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.

Acknowledgements

This research was supported in part by the Agence Nationale de la Recherche (ANR) grant agreement ANR-22-CE39-0014-03 (EMASS project).

Data-Availability Statement

The software that supports §8 is available on Zenodo [71]. 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] 2024. CODEX site. https://codex.top.

[2] Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. Princeton University. http://www.ccs.neu.edu/home/amal/ahmedthesis.pdf.

[3] 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

[4] 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.

[5] Andrea Arcangeli, David Woodhouse, and Michel Lespinasse. 2012. *Linux Kernel Red Black Trees.* https://github.com/torvalds/linux/blob/5133c9e51de41bfa902153888e11add3342ede18/lib/rbtree.c

[6] 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

[7] Thaïs 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

[8] 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.

[9] Rodney M Burstall. 1972. Some techniques for proving correctness of programs which alter data structures. *Machine intelligence* 7, 23-50 (1972), 3. https://doi.org/10.1007/10722010_8

[10] 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. https://doi.org/10.1007/s10817-018-9457-5

[11] Satish Chandra and Thomas Reps. 1999. Physical type checking for C. *ACM SIGSOFT Software Engineering Notes* 24, 5 (1999), 66–75. https://doi.org/10.1145/381788.316183

[12] 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

[13] 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. https://doi.org/10.1145/1993498.1993526

[14] 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

[15] 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

[16] 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

[17] Patrick Cousot. 2021. *Principles of Abstract Interpretation.* MIT Press. https://books.google.fr/books?id=CUoQEAAAQBAJ

[18] 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 1977).* 238–252. https://doi.org/10.1145/512950.512973

[19] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles of Programming Languages (POPL 1979).* 269–282. https://doi.org/10.1145/567752.567778

[20] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, R.N. Horspool (Ed.). LNCS 2304, Springer, Berlin, Grenoble, France, 159–178. https://doi.org/10.5555/647478.727794

[21] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In *Conference Record of the 38$^{\text{th}}$ Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, New York, Austin, Texas, 105–118. https://doi.org/10.1145/1925844.1926399

[22] Luis Manuel Martins Damas. 1984. *Type Assignment in Programming Languages.* Ph. D. Dissertation. University of Edinburgh.

[23] Vincent David, Christophe Aussaguès, Stéphane Louise, Philippe Hilsenkopf, Bertrand Ortolo, and Christophe Hessler. 2004. The oasis based qualified display system. In *Fourth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT 2004), Columbus, Ohio, USA*. 11.

[24] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based Alias Analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98) (PLDI'98)*. ACM, New York, NY, USA, 106–117. https://doi.org/10.1145/277650.277670

[25] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: type and memory error detection using dynamically typed C/C++. In *Proceedings of PLDI*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 181–195. https://doi.org/10.1145/3192366.3192388

[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 in Computer Science, Vol. 7935)*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer, 215–237. https://doi.org/10.1007/978-3-642-38856-9_13

[27] Adam Dunkels. 2023. *Contiki-OS*. https://github.com/contiki-ng/contiki-ng/blob/release-4.9/os/lib/list.c

[28] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*. IEEE Computer Society, 53–60. https://doi.org/10.1109/SecDev.2018.00015

[29] Free Software Foundation. 2022. *GNU Emacs source repository*. https://git.savannah.gnu.org/cgit/emacs.git/tree/lisp.h?h=emacs-28.2

[30] Jacques Garrigue. 2004. Relaxing the Value Restriction. In *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2998)*, Yukiyoshi Kameyama and Peter J. Stuckey (Eds.). Springer, 196–213. https://doi.org/10.1007/978-3-540-24754-8_15

[31] Philippe Granger. 1989. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics* 30, 3-4 (1989), 165–190. https://doi.org/10.1080/00207168908803778

[32] Philippe Granger. 1992. Improving the Results of Static Analyses Programs by Local Decreasing Iteration. In *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, 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

[33] Matthew Harren. 2007. *Dependent Types for Assembly Code Safety.* Ph. D. Dissertation. University of California at Berkeley.

[34] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *NASA Formal Methods* 6617 (2011), 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[35] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[36] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. 2018. Modular static analysis of string manipulations in C programs. In *Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25*. Springer, 243–262. https://doi.org/10.1007/978-3-319-99725-4_16

[37] Stephen Kell. 2016. Dynamically diagnosing type errors in unsafe code. In *Proceedings of OOPSLA*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 800–819. https://doi.org/10.1145/2983990.2983998

[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. https://doi.org/10.1007/978-3-642-11319-2_17

[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. https://doi.org/10.1145/3554341

[43] 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

[44] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2014. *The CompCert memory model.* Cambridge University Press, Chapter 32.

[45] 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

[46] 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. https://doi.org/10.1145/3093333.3009881

[47] 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

[48] 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

[49] 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

[50] 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

[51] 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.

[52] Antoine Miné. 2006. The octagon abstract domain. *Higher-order and symbolic computation* 19, 1 (2006), 31–100. https://doi.org/10.1109/WCRE.2001.957836

[53] 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

[54] 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

[55] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. 1999. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*. 25–35.

[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. https://doi.org/10.1145/319301.319345

[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), 477–526. https://doi.org/10.1145/1065887.1065892

[61] Olivier Nicole, Matthieu Lemerre, and Xavier Rival. 2022. Lightweight Shape Analysis Based on Physical Types. In *23rd International Conference on Verification, Model Checking, and Abstract Interpretation – VMCAI 2022 (Lecture Notes in Computer Science, Vol. 13182)*, Bernd Finkbeiner and Thomas Wies (Eds.). Springer, 219–241. https://doi.org/10.1007/978-3-030-94583-1_11

[62] Jens Palsberg. 2001. Type-based analysis and applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, John Field and Gregor Snelting (Eds.). ACM, 20–27. https://doi.org/10.1145/379605.379635

[63] Marina Polishchuk, Ben Liblit, and Chloë W. Schulze. 2007. Dynamic heap type inference for program understanding and debugging. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 39–46. https://doi.org/10.1145/1190216.1190225

[64] Reese T. Prosser. 1959. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference* (Boston, Massachusetts) *(IRE-AIEE-ACM '59 (Eastern))*. ACM, New York, NY, USA, 133–138. https://doi.org/10.1145/1460299.1460314

[65] John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on.* IEEE, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[66] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. 1995. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.* 17 (1995), 233–263. https://doi.org/10.1145/201059.201065

[67] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *POPL*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/1706299.1706316

[68] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally with Checked C. In *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11426)*, Flemming Nielson and David Sands (Eds.). Springer, 76–98. https://doi.org/10.1007/978-3-030-17138-4_4

[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 SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. https://doi.org/10.1145/3453483.3454036

[70] Julien Simonnet, Matthieu Lemerre, and Mihaela Sighireanu. 2024. *A dependent nominal physical type system for the static analysis of memory in low level code.* Technical Report. https://hal.science/hal-04649674 Full version with appendices.

[71] Julien Simonnet, Matthieu Lemerre, and Mihaela Sighireanu. 2024. Artifact for the paper A Dependent Nominal Physical Type System for the Static Analysis of Memory in Low Level Code. https://doi.org/10.5281/zenodo.13383433

[72] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. 48–62. https://doi.org/10.1109/SP.2013.13

[73] Robert D. Tennent and Dan R. Ghica. 2000. Abstract Models of Storage. *High. Order Symb. Comput.* 13, 1/2 (2000), 119–129. https://doi.org/10.1023/A:1010022312623

[74] Mads Tofte. 1990. Type Inference for Polymorphic References. *Inf. Comput.* 89, 1 (1990), 1–34. https://doi.org/10.1016/0890-5401(90)90018-D

[75] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 684–714. https://doi.org/10.1007/978-3-030-44914-8_25

[76] Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, bytes, and separation logic. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 97–108. https://doi.org/10.1145/1190216.1190234

[77] David Walker. 2005. *Advanced topics in types and programming languages.* The MIT Press Cambridge, Chapter Substructural type systems, 3–44.

[78] Andrew K. Wright. 1995. Simple Imperative Polymorphism. *LISP Symb. Comput.* 8, 4 (1995), 343–355. https://doi.org/10.1007/BF01018828

[79] Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and computation* (1994). https://doi.org/10.1006/inco.1994.1093

[80] Hongwei Xi and Robert Harper. 2001. A Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 169–180. https://doi.org/10.1145/507635.507657

[81] 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.

A Physical Dependent Types

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

A.1 Well-formed Type Definitions

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

Proposition 1. *A well-founded relation $\prec_\Delta$ induces a DAG over type expressions and a domination relation $\sqsubseteq$ whose entry is* `byte`.

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

The domination relation [4, 64] is defined by $\tau \sqsubseteq \tau'$ iff all paths in the DAG of $\prec_\Delta$ from the entry `byte` to τ are included in the set of paths from the entry to τ'. □

A.2 Lattice of Type-offsets

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

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

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

Proof. By induction on the definition of $\prec_{\mathbb{P}_\mathbb{m}}$ in Fig. 5 using the definition of $\prec_\Delta$ in Table 2. □

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

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

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

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

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

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

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

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

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

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

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

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

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

(1) $\gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_1, k_1) \subseteq \gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_2, k_2)$ *iff* $(\tau_1, k_1) \sqsubseteq_{\mathbb{P}_{\mathbb{m}}} (\tau_2, k_2)$, *i.e., the domination relation corresponds to inclusion of type-offset concretizations into well-formed sequences of TO;*

(2) $\exists (\tau, k) \in \mathbb{P}_{\mathbb{m}}$ *such that* $\gamma_{\mathbb{P}^+,\mathbb{m}}(\tau, k) = \gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_1, k_1) \cup \gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_2, k_2)$ *iff* $(\tau, k) = (\tau_1, k_1) \sqcup_{\mathbb{P}_{\mathbb{m}}} (\tau_2, k_2)$, *i.e., the join operator computes the least common dominator.*

(3) $\gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_1, k_1) \cap \gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_2, k_2) = \emptyset$ *iff there is no* (τ, k) *such that* $(\tau, k) \prec^*_{\mathbb{P}_{\mathbb{m}}} (\tau_1, k_1)$ *and* $(\tau, k) \prec^*_{\mathbb{P}_{\mathbb{m}}} (\tau_2, k_2)$.

Proof. Concretization $\gamma_{\mathbb{P}^+,\mathbb{m}}$ captures all the paths traversing a type-offset. □

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

(1) *If* $(\tau_1, k_1) \sqsubseteq_{\mathbb{P},\mathbb{m}} (\tau_2, k_2)$ *then* $\gamma_{\mathbb{P},\mathbb{m}}(\tau_1, k_1) \supseteq \gamma_{\mathbb{P},\mathbb{m}}(\tau_2, k_2)$, *i.e., a dominating type-offset includes the addresses of the dominated type-offset.*

(2) *If* $\gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_1, k_1) \cap \gamma_{\mathbb{P}^+,\mathbb{m}}(\tau_2, k_2) = \emptyset$ *then* $\gamma_{\mathbb{P},\mathbb{m}}(\tau_1, k_1) \cap \gamma_{\mathbb{P},\mathbb{m}}(\tau_2, k_2) = \emptyset$, *i.e., if there is no path going through both* (τ_1, k_1) *and* (τ_2, k_2) *then they don't share addresses.*

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

(2) Same reasoning as above.

□

B A Low Level Programming Language

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

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

$$\begin{array}{llcl}
\text{Expressions} & \mathsf{Exp} \ni E & ::= & k \mid x \mid E \diamond E \mid *_{\ell} E \\
\text{Commands} & \mathsf{Cmd} \ni C & ::= & \mathbf{skip} \mid x := E \mid x := \mathbf{malloc}_{\eta}(E) \mid *_{\ell} E := E \mid \mathbf{assume}\ E \\
 & & \mid & C ; C \mid \mathbf{while}\ E\ \mathbf{do}\ C \mid \mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ C
\end{array}$$

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

B.1 Syntax

The syntax of While$_{\text{MEM}}$ is given by the grammar in Fig. 7. Memory locations are obtained using dynamic memory allocation and pointer arithmetics. We denote by $\mathbb{A}$ the set of addresses, i.e., values of memory locations. To simplify the presentation, we consider a little-endian ABI, although the endianness is a parameter of our analysis. The values $\mathbb{V}$ are bit vectors, i.e., non-negative integers represented as fixed size sequences of bytes. The arithmetic, logical and comparison operators $\diamond \in \{+, -, \times, /, \&, |, =, \leq, \ldots\}$ are extended to bit vector values in $\mathbb{V}$ using the semantic given by the ABI in a classic way. Like in C or assembler, we interpret non null expressions as true and null expressions as false. Comparison expressions evaluate to 1 when they hold and to 0 otherwise. The memory may be read (resp. written) on ℓ bytes at an address given by an expression E using the load expression $*_\ell E$ (resp. store command $*_\ell E := E'$). Due to pointer arithmetics allowed in expressions, the grammar of expressions is enough to encode array access $E[E']$ or field access $E.E'$. We assume that instances of **malloc** are labeled by type names η defined in the program and allocates a number of bytes equal to E times the size of η. The **assume** command always succeeds and changes the program's state such that the expression in argument becomes true; it is used to insert annotations in the program's abstract state.

B.2 Untyped Semantics

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

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

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

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

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

Dynamic semantics. To define the execution semantics of our programming language, we introduce the notions of store and heap.

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

$$\sigma \in \Sigma \triangleq [\mathbb{X} \longrightarrow \mathbb{V}]$$

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 mapped by σ in x.

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

$$\mathrm{h} \in \mathbb{H} \triangleq [\mathbb{A} \rightharpoonup \mathbb{V}_1]$$

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

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

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

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

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

The semantics of commands is defined by the judgment $s \vdash C \Rightarrow s'$ meaning that the command 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.

We comment some of these rules.

$$\frac{}{(\sigma, \mathrm{h}) \ \vdash \ x \Downarrow \sigma[x]} \ \textsc{Env} \qquad\qquad \frac{}{s \ \vdash \ k \Downarrow k} \ \textsc{Const}$$

$$\frac{s \ \vdash \ E \Downarrow a \qquad [a, a+\ell] \subseteq \mathrm{dom}(\mathrm{h})}{s \ \vdash \ *_\ell E \Downarrow \mathrm{h}[a..a+\ell]} \ \textsc{Load}$$

$$\frac{s \ \vdash \ E_1 \Downarrow v_1 \qquad s \ \vdash \ E_2 \Downarrow v_2 \qquad v_1, v_2 \in \mathbb{V}_\ell \qquad v_2 \neq 0}{s \ \vdash \ E_1 / E_2 \Downarrow v_1 / v_2} \ \textsc{Div}$$

$$\frac{s \vdash E_1 \Downarrow v_1 \qquad s \vdash E_2 \Downarrow v_2 \qquad v_1, v_2 \in \mathbb{V}_\ell \qquad \diamond \in \{+, -, \times, \dots\}}{s \ \vdash \ E_1 \diamond E_2 \Downarrow v_1 \diamond v_2} \ \textsc{Binop}$$

$$\frac{s \ \vdash \ E \Downarrow v \qquad v \in \mathbb{V}_\ell \qquad \triangleright \in \{-, \neg\}}{s \ \vdash \ \triangleright E \Downarrow \triangleright v} \ \textsc{Unop}$$

Fig. 8. Dynamic semantics of expressions in $\textsc{While}_{\textsc{mem}}$

$$\textsc{Skip} \frac{}{s \vdash \mathbf{skip} \Rightarrow s} \qquad\qquad \textsc{Seq} \frac{s \vdash C_1 \Rightarrow s_1 \qquad s_1 \vdash C_2 \Rightarrow s_2}{s \vdash C_1; C_2 \Rightarrow s_2}$$

$$\textsc{Assign} \frac{(\sigma, \mathrm{h}) \ \vdash \ E \Downarrow v}{(\sigma, \mathrm{h}) \vdash x := E \Rightarrow (\sigma[x \leftarrow v], \mathrm{h})}$$

$$\textsc{Store} \frac{(\sigma, \mathrm{h}) \ \vdash \ E_1 \Downarrow a \in \mathbb{V}_{\mathcal{W}} \qquad (\sigma, \mathrm{h}) \ \vdash \ E_2 \Downarrow v \in \mathbb{V}_l}{(\sigma, \mathrm{h}) \vdash \ *_\ell \ E_1 := E_2 \Rightarrow (\sigma, \mathrm{h}[a..a+\ell \leftarrow v])}$$

$$\textsc{Alloc} \frac{(\sigma, \mathrm{h}) \ \vdash \ E \Downarrow \ell \qquad \ell > 0 \qquad [a..a+\ell] \cup \mathrm{dom}(\mathrm{h}) = \varnothing \qquad v \in \mathbb{V}_\ell}{(\sigma, \mathrm{h}) \vdash x := \mathbf{malloc}_\eta(E) \Rightarrow (\sigma[x \leftarrow a], \mathrm{h}[a..a+\ell \leftarrow v])}$$

$$\textsc{Alloc0} \frac{(\sigma, \mathrm{h}) \ \vdash \ E \Downarrow \ell \qquad \ell > 0}{(\sigma, \mathrm{h}) \vdash x := \mathbf{malloc}_\eta(E) \Rightarrow (\sigma[x \leftarrow 0], \mathrm{h})} \qquad \textsc{Assume} \frac{s \ \vdash \ E \Downarrow v \qquad v \neq 0}{s \vdash \mathbf{assume}\ E \Rightarrow s}$$

$$\textsc{Then} \frac{s \vdash \mathbf{assume}\ E; C_1 \Rightarrow s'}{s \vdash \mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{end} \Rightarrow s'} \qquad \textsc{Else} \frac{s \vdash \mathbf{assume}\ \neg E; C_2 \Rightarrow s'}{s \vdash \mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{end} \Rightarrow s'}$$

$$\textsc{WhileEnd} \frac{s \ \vdash \ E \Downarrow 0}{s \vdash \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{done} \Rightarrow s}$$

$$\textsc{WhileLoop} \frac{s \ \vdash \ E \Downarrow v \qquad v \neq 0 \qquad s \vdash C \Rightarrow s_1 \qquad s_1 \vdash \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{done} \Rightarrow s_2}{s \vdash \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{done} \Rightarrow s_2}$$

Fig. 9. Dynamic semantics of statements in $\textsc{While}_{\textsc{mem}}$

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 $\mathbb{A}$ that is in the domain of h), stores the result of evaluating E_2 at that address, if E_2 evaluates to a bit vector size ℓ.

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

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

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

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

B.3 Typed Concrete Semantics

This is an additional material for §6.

Theorem 6.2 (Safe store inside a region). *Let* $(\mathbb{m}, h)$ *be a well-typed heap,* $a \in \mathrm{dom}(h)$ *an allocated address such that* $(\rho, k) \in \overline{\mathbb{m}}(a)$ *and* $s = \mathrm{size}(\rho)$*. 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.*

Proof. Follows from $h[a..a+\ell \leftarrow v] = h[a-k..a-k+s \leftarrow h[a-k..a] :: v :: h[a+\ell..a-k+s]]$. □

Theorem 6.2 (Name matching implies monotony). *Let* $\mathbb{m}, \mathbb{m}' \in \mathbb{A} \rightharpoonup \mathbb{R}$ *such that* $\mathbb{m} \rightsquigarrow \mathbb{m}'$*. Then:* $\forall \tau \in \mathbb{T} : (\!|\, \tau \,|\!)_{\mathbb{m}} \subseteq (\!|\, \tau \,|\!)_{\mathbb{m}'}$ *and* $[\![\, \tau \,]\!]_{\mathbb{m}} \subseteq [\![\, \tau \,]\!]_{\mathbb{m}'}$.

Proof. By structural induction on types using the well-founded relation $<_\Delta$ and the definition of $(\!|\, \cdot \,|\!)_{\mathbb{m}}$ and $[\![\, \cdot \,]\!]_{\mathbb{m}}$ in Table 4. □

C Analysis Rules

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

C.1 Rules's Overview and Their Soundness Theorems

As usual in abstract interpretation [19], our abstract transformers (analysis rules) derive from the definition of our domain and its concretization. These transformers involve many different operations requiring interaction between the different parts of the analysis. To simplify the presentation, we describe the abstract transformers as a set of deduction rules over the formulas that our abstract domains represent. These formulas are given as judgements below. The rules could be used as a proof witness of the type safety of the program.

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

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

- Rules that check the safety of stores to the heap $\{s_1^\sharp\} \ *_\ell\ \hat{e}_1 := \hat{e}_2\ \{s_2^\sharp\}$;
- Rules for joining states $s_1^\sharp \sqcup s_2^\sharp$ 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 $s^\sharp, s_1^\sharp s_2^\sharp \in \mathbb{S}^\sharp$; 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 $\mathbb{m}$ and ν:*

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

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

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

Numerical queries *If $s^\sharp \vDash \hat{e}$, $s \in \gamma_{\mathbb{S}^\sharp}(s^\sharp)$ and $s \in \gamma_{\mathbb{M}^\sharp}(s^\sharp)(\mathbb{m}, \nu)$, then $\gamma_{\hat{\mathbb{E}}}(\hat{e})(\nu) \neq 0$.*

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

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

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

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

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

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 $(\hat{\tau}_1, \hat{e}_1) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}_2, \hat{e}_2)$ to mean that given any valuation ν, the corresponding (τ_1, k_1) and (τ_2, k_2) that the symbolic type represent are such that $(\tau_1, k_1) \sqsubseteq_{\mathbb{P}}^\sharp (\tau_2, k_2)$. However, this could be too imprecise, we actually want to consider only the valuations that can be represented by a given state $s^\sharp$. Hence, our judgments are of the form $s^\sharp \vDash (\hat{\tau}_1, \hat{e}_1) \sqsubseteq_{\mathbb{P}}^\sharp (\hat{\tau}_2, \hat{e}_2)$.

$$\textsc{LeqByte}\quad s^\sharp \models (\hat{\tau}, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\texttt{byte}[\text{size}(\hat{\tau})], \hat{e}) \qquad \textsc{LeqNamed}\quad s^\sharp \models (\hat{\eta}, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\Delta(\hat{\eta}), \hat{e}) \qquad \textsc{LeqWith}\quad s^\sharp \models (\{x : \hat{\tau} \mid \hat{e}_1\}, \hat{e}_2) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}, \hat{e}_2)$$

$$\textsc{LeqInter}\ \frac{s^\sharp \models (\hat{\tau}_2, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}_3, \hat{e}_3)}{s^\sharp \models (\hat{\tau}_1 \wedge \hat{\tau}_2, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}_3, \hat{e}_3)} \qquad \textsc{LeqUnion}\ \frac{\begin{array}{c} s^\sharp \models (\hat{\tau}_1, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}, \hat{e}_3) \\ s^\sharp \models (\hat{\tau}_2, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}, \hat{e}_3) \end{array}}{s^\sharp \models (\hat{\tau}_1 \cup \hat{\tau}_2, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}, \hat{e}_3)}$$

$$\textsc{LeqProd}\ \frac{\begin{array}{cc} s^\sharp \wedge \hat{e} \leq \text{size}(\hat{\tau}_1) \Rightarrow s^\sharp_1 & s^\sharp_1 \models (\hat{\tau}_1, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}_3, \hat{e}_3) \\ s^\sharp \wedge \hat{e} > \text{size}(\hat{\tau}_1) \Rightarrow s^\sharp_2 & s^\sharp_2 \models (\hat{\tau}_2, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}_3, \hat{e}_3) \end{array}}{s^\sharp \models (\hat{\tau}_1 \times \hat{\tau}_2, \hat{e}) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}_3, \hat{e}_3)}$$

$$\textsc{LeqArray}\ \frac{s^\sharp \models \text{size}(\hat{\tau}) = \ell \qquad s^\sharp \models 0 \leq \hat{e}_2 < \text{size}(\ell * \hat{e}) \qquad s^\sharp \models (\hat{\tau}, \hat{e} \bmod \ell) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}_3, \hat{e}_3)}{s^\sharp \models (\hat{\tau}[\hat{e}], \hat{e}_2) \sqsubseteq^\sharp_\mathbb{P} (\hat{\tau}_3, \hat{e}_3)}$$

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 $\bot$, 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 though 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 v, but only those that correspond to the states $s^\sharp_1$ and $s^\sharp_2$ 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 $int[4] + 1$ in $s^\sharp_1$, and by $int[7] + 4$ in

$s_2^\sharp$, then it will be in a set $int[\alpha_1] + \alpha_2$ 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 [12] $\phi \in \hat{\mathbb{E}} \times \hat{\mathbb{E}} \rightharpoonup \mathbb{\alpha}$ takes a pair of different expression and returns a fresh (or deterministically named [42]) 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* $\mathrm{subst}(\hat{\tau}_1, \phi_1) = \phi$ *and* $\mathrm{subst}(\hat{\tau}_2, \phi_2) = \hat{\tau}$.

MatchingSame

$$\vDash_\phi \phi(\hat{\tau}, \hat{\tau}) = \hat{\tau}$$

MatchingName

$$\vDash_\phi \phi(\mathtt{n}(\hat{e}_1, \ldots, \hat{e}_\ell), \mathtt{n}(\hat{e}_{\ell+1}, \ldots, \hat{e}_{2\ell})) = \mathtt{n}(\phi(\hat{e}_1, \hat{e}_{\ell+1}), \ldots, \phi(\hat{e}_\ell, \hat{e}_{2\ell}))$$

MatchingWith

$$\frac{\vDash_\phi \phi(\hat{\tau}_1, \hat{\tau}_2) = \hat{\tau}}{\vDash_\phi \phi(\{x : \hat{\tau}_1 \mid \hat{e}_1\}, \{x : \hat{\tau}_2 \mid \hat{e}_2\}) = \{x : \hat{\tau} \mid \phi(\hat{e}_1, \hat{e}_2)\}}$$

MatchingProd

$$\frac{\vDash_\phi \phi(\hat{\tau}_{11}, \hat{\tau}_{12}) = \hat{\tau}_1 \qquad \vDash_\phi \phi(\hat{\tau}_{21}, \hat{\tau}_{22}) = \hat{\tau}_2 \qquad \mathrm{size}(\hat{\tau}_{11}) = \mathrm{size}(\hat{\tau}_{12})}{\vDash_\phi \phi(\hat{\tau}_{11} \times \hat{\tau}_{21}, \hat{\tau}_{12} \times \hat{\tau}_{22}) = \hat{\tau}_1 \times \hat{\tau}_2}$$

MatchingArray

$$\frac{\vDash_\phi \phi(\hat{\tau}_1, \hat{\tau}_2) = \hat{\tau} \qquad \mathrm{size}(\hat{\tau}_1) = \mathrm{size}(\hat{\tau}_2)}{\phi(\hat{\tau}_1\,\mathtt{[}\hat{e}_1\mathtt{]}, \hat{\tau}_2\,\mathtt{[}\hat{e}_2\mathtt{]}) = \hat{\tau}\,\mathtt{[}\phi(\hat{e}_1, \hat{e}_2)\mathtt{]}}$$

MatchingUnion

$$\frac{\vDash_\phi \phi(\hat{\tau}_{11}, \hat{\tau}_{12}) = \hat{\tau}_1 \qquad \vDash_\phi \phi(\hat{\tau}_{21}, \hat{\tau}_{22}) = \hat{\tau}_2}{\vDash_\phi \phi(\hat{\tau}_{11} \cup \hat{\tau}_{21}, \hat{\tau}_{12} \cup \hat{\tau}_{22}) = \hat{\tau}_1 \cup \hat{\tau}_2}$$

MatchingInter

$$\frac{\vDash_\phi \phi(\hat{\tau}_{11}, \hat{\tau}_{12}) = \hat{\tau}_1 \qquad \vDash_\phi \phi(\hat{\tau}_{21}, \hat{\tau}_{22}) = \hat{\tau}_2}{\vDash_\phi \phi(\hat{\tau}_{11} \wedge \hat{\tau}_{21}, \hat{\tau}_{12} \wedge \hat{\tau}_{22}) = \hat{\tau}_1 \wedge \hat{\tau}_2}$$

MatchingExists

$$\frac{\vDash_\phi \phi(\hat{\tau}_{11}, \hat{\tau}_{12}) = \hat{\tau}_1 \qquad \vDash_\phi \phi(\hat{\tau}_{21}, \hat{\tau}_{22}) = \hat{\tau}_2}{\vDash_\phi \phi(\exists \alpha : \hat{\tau}_{11}.\, \hat{\tau}_{21}, \exists \alpha : \hat{\tau}_{12}.\, \hat{\tau}_{22}) = \exists \alpha : \hat{\tau}_1.\, \hat{\tau}_2}$$

MatchingAddr

$$\frac{\vDash_\phi \phi(\hat{\tau}_1, \hat{\tau}_2) = \hat{\tau}}{\vDash_\phi \phi(\hat{\tau}_1\star + \hat{e}_1, \hat{\tau}_2\star + \hat{e}_2) = \hat{\tau}\star + \phi(\hat{e}_1, \hat{e}_2)}$$

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 $\mathrm{dom}(\phi)$:

$$\textsc{JoingJudgments}\quad \frac{\begin{array}{cc} s_1^\sharp \vDash \hat{e}_1 : \hat{\tau}_{11}\star + \hat{e}_{11} & s_1^\sharp \vDash (\hat{\tau}_{11}, \hat{e}_{11}) \sqsubseteq^\sharp_{\mathbb{P}} (\hat{\tau}_{111}, \hat{e}_{111}) \\ s_2^\sharp \vDash \hat{e}_2 : \hat{\tau}_{22}\star + \hat{e}_{22} & s_2^\sharp \vDash (\hat{\tau}_{22}, \hat{e}_{22}) \sqsubseteq^\sharp_{\mathbb{P}} (\hat{\tau}_{222}, \hat{e}_{222}) \\ \multicolumn{2}{c}{\vDash_\phi \phi(\hat{\tau}_{111}, \hat{\tau}_{222}) = \hat{\tau}} \end{array}}{s_1^\sharp \sqcup s_2^\sharp \vDash \phi(\hat{e}_1, \hat{e}_2) : (\hat{\tau}\star + \phi(\hat{e}_{111}, \hat{e}_{222}))}$$

- **JoinJudgments** explains how $\Gamma^\sharp$ is computed: for each pair $(\hat{e}_1, \hat{e}_2)$ of expressions which needs to be joined, we get their respective values $s_1^\sharp.\Gamma^\sharp[\hat{e}_1]$ and $s_2^\sharp.\Gamma^\sharp[\hat{e}_2]$, climb the $\sqsubseteq^\sharp_{\mathbb{P}}$ 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 $\Gamma^\sharp$, in the case where these expressions could represent some adresses).

We define $\sqcup^\phi_{\Gamma^\sharp}$ as the smallest $\Gamma^\sharp$ that can produce these judgments (given a ϕ).

We suppose that $\sqcup^\phi_{\mathbb{V}^\sharp}$ is given, and $\sqcup^\phi_{\Sigma^\sharp}$ is defined as follows:

$$\sigma_1^\sharp \sqcup^\phi_{\Sigma^\sharp} \sigma_2^\sharp \triangleq \lambda x \in \mathbb{X}.\ \phi(\sigma_1^\sharp[x], \sigma_2^\sharp[x])$$

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

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

where *minimal* means that $\mathrm{dom}(\phi)$ 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 $s_1^\sharp, s_2^\sharp \in \mathbb{S}^\sharp$:*

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

.

C.4 Rules for Compound Commands

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

SKIP

$$\{s^\sharp\}\ \texttt{skip}\ \{s^\sharp\}$$

SEQ

$$\frac{\{s_0^\sharp\}\ C_1\ \{s_1^\sharp\} \qquad \{s_1^\sharp\}\ C_2\ \{s_2^\sharp\}}{\{s_0^\sharp\}\ C_1; C_2\ \{s_2^\sharp\}}$$

IF

$$\frac{\{s_0^\sharp\}\ \texttt{assume}(E); C_1\ \{s_1^\sharp\} \qquad \{s_0^\sharp\}\ \texttt{assume}(\neg E); C_2\ \{s_2^\sharp\} \qquad s_3^\sharp = s_1^\sharp \sqcup^\sharp s_2^\sharp}{\{s_0^\sharp\}\ \texttt{if}(E)\ C_1\ \texttt{else}\ C_2\ \{s_3^\sharp\}}$$

WHILE-DONE

$$\frac{\{s_0^\sharp\}\ \texttt{assume}(e); C_1\ \{s_1^\sharp\} \qquad s_1^\sharp \sqsubseteq s_0^\sharp \qquad \{s_1^\sharp\}\ \texttt{assume}(\neg E)\ \{s_2^\sharp\}}{\{s_0^\sharp\}\ \texttt{while}(E)\ \texttt{do}\ C_1\ \texttt{done}\ \{s_2^\sharp\}}$$

WHILE-AGAIN

$$\frac{\{s_0^\sharp\}\ \texttt{assume}(E); C_1\ \{s_1^\sharp\} \qquad \{s_0^\sharp \nabla s_1^\sharp\}\ \texttt{while}(E)\ \texttt{do}\ C_1\ \texttt{done}\ \{s_2^\sharp\}}{\{s_0^\sharp\}\ \texttt{while}(E)\ \texttt{do}\ C_1\ \texttt{done}\ \{s_2^\sharp\}}$$

C.5 Rules for Basic Commands

ASSIGN

$$\frac{\{s_0^\sharp\}\ E \Downarrow \hat{e}\ \{s_1^\sharp\}}{\{s_0^\sharp\}\ x := E\ \{s_2^\sharp[\sigma^\sharp \leftarrow s_1^\sharp.\sigma^\sharp[x \leftarrow \hat{e}]]\}}$$

ASSUME

$$\frac{\{s_0^\sharp\}\ E \Downarrow \hat{e}\ \{s_1^\sharp\} \qquad v_2^\sharp = s_1^\sharp.v^\sharp \wedge (\hat{e} \neq 0) \qquad s_1^\sharp[v^\sharp \leftarrow v_2^\sharp] \Rightarrow s_3^\sharp}{\{s_0^\sharp\}\ \texttt{assume}(E)\ \{s_3^\sharp\}}$$

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

VAR

$$\{s^\sharp\}\ x \Downarrow s^\sharp.\sigma^\sharp[x]\ \{s^\sharp\}$$

LOAD

$$\frac{\{s_0^\sharp\}\ E \Downarrow \hat{e}_1\ \{s_1^\sharp\} \qquad \{s_1^\sharp\}\ \star_\ell \hat{e}_1 \Downarrow \hat{e}_2\ \{s_2^\sharp\}}{\{s_0^\sharp\}\ \star_\ell E \Downarrow \hat{e}_2\ \{s_2^\sharp\}}$$

CONST

$$\{s^\sharp\}\ k \Downarrow k\ \{s^\sharp\}$$

BINOP

$$\frac{\{s_0^\sharp\}\ E_1 \Downarrow \hat{e}_1\ \{s_1^\sharp\} \qquad \{s_1^\sharp\}\ E_2 \Downarrow \hat{e}_2\ \{s_2^\sharp\}}{\{s^\sharp\}\ E_1 \diamond E_2 \Downarrow \hat{e}_1 \diamond \hat{e}_2\ \{s_2^\sharp\}}$$

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 $s_2^\sharp \models \hat{e}_2 \neq 0$ to the rule Binop when the binary operator $\diamond$ is the division.

C.7 The Size Operator on Abstract Types

$$
\begin{aligned}
\text{size} &: \hat{\mathbb{T}} \rightharpoonup \hat{\mathbb{E}} \\
\text{size}(\texttt{byte}) &\triangleq 1 \\
\text{size}(\mathsf{n}(\hat{e}_1, \ldots, \hat{e}_\ell)) &\triangleq \text{size}(\Delta(\mathsf{n}(\hat{e}_1, \ldots, \hat{e}_\ell))) \\
\text{size}(\eta\star + \hat{e}) &\triangleq \mathcal{W} \\
\text{size}(\{x : \hat{\tau} \mid \hat{e}\}) &\triangleq \text{size}(\hat{\tau}) \\
\text{size}(\hat{\tau}_1 \times \hat{\tau}_2) &\triangleq \text{size}(\hat{\tau}_1) + \text{size}(\hat{\tau}_2) \\
\text{size}(\hat{\tau}\,\texttt{[}\hat{e}\texttt{]}) &\triangleq \hat{e} * \text{size}(\hat{\tau}) \\
\text{size}(\hat{\tau}_1 \cup \hat{\tau}_2) &\triangleq \text{size}(\hat{\tau}_1) \text{ if } \text{size}(\hat{\tau}_1) = \text{size}(\hat{\tau}_2) \\
\text{size}(\exists \alpha : \hat{\tau}_1.\, \hat{\tau}_2) &\triangleq \text{size}(\hat{\tau}_2) \text{ if } \alpha \notin \text{size}(\hat{\tau}_2) \\
\text{size}(\hat{\tau}_1 \bigwedge \hat{\tau}_2) &\triangleq \text{size}(\hat{\tau}_1)
\end{aligned}
$$

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.

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

$$
\textsc{LoadLarger}\quad
\frac{\begin{array}{c} s_0^\# \vDash \hat{e}_1 : (\hat{\tau}\star + \hat{e}_2) \\ s_0^\# \vDash \hat{e}_2 = k \qquad s_0^\# \vDash \text{size}(\hat{\tau}) = \ell_2 \qquad \ell_2 \geq k + \ell \qquad \{s_0^\#\} \star_{\ell_2} (\hat{e}_1 - k) \Downarrow \hat{e}_3\ \{s_1^\#\} \end{array}}{\{s_0^\#\} \star_{\ell_1} \hat{e}_1 \Downarrow \hat{e}_3[k..k + \ell_1]\ \{s_1^\#\}}
$$

$$
\textsc{LoadUnionSplit}\quad
\frac{\begin{array}{c} s_0^\# \vDash \hat{e} : \hat{\tau}_1 \cup \hat{\tau}_2 \\ s_0^\# \wedge \hat{e} : \hat{\tau}_1 \Rightarrow s_1^\# \qquad \{s_1^\#\} \star_\ell \hat{e} \Downarrow \hat{e}_1\ \{s_1'^\#\} \qquad s_0^\# \wedge \hat{e} : \hat{\tau}_2 \Rightarrow s_2^\# \qquad \{s_2^\#\} \star_\ell \hat{e} \Downarrow \hat{e}_2\ \{s_2'^\#\} \end{array}}{\{s_0^\#\} \star_\ell \hat{e} \Downarrow \phi(\hat{e}_1, \hat{e}_2)\ \{s_1'^\# \sqcup s_2'^\#\}}
$$

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 $\hat{\tau}_1 \bigwedge \hat{\tau}_2$), 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 PtrClimbAddrLattice, 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 PtrClimbAddrLattice 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

$$\frac{\begin{array}{c} s^\sharp \vDash \hat{e}_1 : (\hat{\eta}\star + \hat{e}_2) \qquad s^\sharp \vDash \mathrm{size}(\hat{\tau}) = \ell \\ s^\sharp \vDash \hat{e}_2 = 0 \qquad \nexists(\hat{\eta}_3, \hat{e}_3) \text{ s.t. } \hat{\eta}_3 \neq \hat{\eta} \wedge (\hat{\eta}_3, \hat{e}_3) \sqsubseteq^\sharp_{\mathbb{P}} (\hat{\eta}, \hat{e}_2) \qquad \hat{e}_4 : \hat{\eta} \qquad \big(s^\sharp \wedge (\alpha : \hat{\tau})\big) \Rightarrow s^\sharp_1 \end{array}}{\{s^\sharp\}\ \star_\ell \hat{e}_1 := \hat{e}_4\ \{s^\sharp\}}$$

StoreLarger

$$\frac{\begin{array}{c} s^\sharp_0 \vDash \hat{e}_1 : (\hat{\tau}\star + \hat{e}_2) \qquad s^\sharp_0 \vDash \hat{e}_2 = k \qquad s^\sharp_0 \vDash \mathrm{size}(\hat{\tau}) = \ell_2 \\ \{s^\sharp_0\}\ \star_{\ell_2}(\hat{e}_1 - k) \Downarrow \hat{e}_3\ \{s^\sharp_1\} \qquad \{s^\sharp_0\}\ \star_{\ell_2}(\hat{e}_1 - k) := \hat{e}_3[0..k] :: \hat{e}_4 :: \hat{e}_3[k + \ell_1..\ell_2]\ \{s^\sharp_1\} \end{array}}{\{s^\sharp_0\}\ \star_{\ell_1} \hat{e}_1 := \hat{e}_4\ \{s^\sharp_1\}}$$

StoreUnionSplit

$$\frac{\begin{array}{c} s^\sharp_0 \vDash \hat{e} : \hat{\tau}_1 \cup \hat{\tau}_2 \\ s^\sharp_0 \wedge \hat{e} : \hat{\tau}_1 \Rightarrow s^\sharp_1 \qquad \{s^\sharp_1\}\ \star_\ell \hat{e} \Downarrow \hat{e}_2\ \{s'^\sharp_1\} \\ s^\sharp_0 \wedge \hat{e} : \hat{\tau}_2 \Rightarrow s^\sharp_2 \qquad \{s^\sharp_2\}\ \star_\ell \hat{e} \Downarrow \hat{e}_2\ \{s'^\sharp_2\} \end{array}}{\{s^\sharp_0\}\ \star_\ell \hat{e} := \hat{e}_2\ \{s'^\sharp_1 \sqcup s'^\sharp_2\}}$$

- **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

PtrCombineInfo

$$\frac{s^\sharp \vDash \hat{e}_1 : \hat{\tau}_1\star + \hat{e}_2 \qquad s^\sharp \vDash \hat{e}_1 : \hat{\tau}_2\star + \hat{e}_2}{s^\sharp \vDash \hat{e}_1 : ((\hat{\tau}_1 \wedge \hat{\tau}_2)\star + \hat{e}_2)}$$

PtrClimbAddrLattice

$$\frac{s^\sharp \vDash \hat{e} : (\hat{\tau}_1\star + \hat{e}_1) \qquad s^\sharp \vDash (\hat{\tau}_1, \hat{e}_1) \sqsubseteq^\sharp_{\mathbb{P}} (\hat{\tau}_2, \hat{e}_2) \qquad s^\sharp \vDash 0 \leq \hat{e}_1 \leq \mathrm{size}(\hat{\tau}_1)}{s^\sharp \vDash \hat{e} : (\hat{\tau}_2\star + \hat{e}_2)}$$

PtrAdd

$$\frac{s^\sharp \vDash \hat{e} : (\hat{\tau}\star + \hat{e}_1)}{s^\sharp \vDash (\hat{e} + \hat{e}_2) : \hat{\tau}\star + (\hat{e}_1 + \hat{e}_2)}$$

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 $\hat{e}_1 : \{x : \mathtt{byte} \mid x \leq 14\}\star$ 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

UseGamma

$$\frac{s^\sharp.\Gamma^\sharp[\hat{e}_2] = (\hat{\tau}, \hat{e})}{s^\sharp \vDash \hat{e}_2 : (\hat{\tau}\star + \hat{e})}$$

UpcastName

$$\frac{s^\sharp \vDash \hat{e} : \hat{\eta}}{s^\sharp \vDash \hat{e} : \Delta(\hat{\eta})}$$

UpcastWith

$$\frac{s^\sharp \vDash \hat{e}_1 : \{x : \hat{\tau} \mid \hat{e}_2\}}{s^\sharp \vDash \hat{e}_1 : \hat{\tau}}$$

PropagateWith

$$\frac{s^\sharp \vDash \hat{e}_1 : (\{x : \hat{\tau} \mid \hat{e}_2\})}{s^\sharp \vDash \hat{e}_2 \neq 0}$$

UpCastProd1

$$\frac{s^\sharp \vDash \hat{e} : (\hat{\tau}_1 \times \hat{\tau}_2) \qquad \mathrm{size}(\hat{\tau}_1) = \ell}{s^\sharp \vDash \hat{e}[0..\ell] : \hat{\tau}_1}$$

UpCastProd2

$$\frac{s^\sharp \vDash \hat{e} : (\hat{\tau}_1 \times \hat{\tau}_2) \qquad \mathrm{size}(\hat{\tau}_1) = \ell_1 \qquad \mathrm{size}(\hat{\tau}_1 \times \hat{\tau}_2) = \ell}{s^\sharp \vDash \hat{e}[\ell_1..\ell] : \hat{\tau}_2}$$

UpCastUnion

$$\frac{s^\sharp \vDash \hat{e} : (\hat{\tau}_1 \cup \hat{\tau}_2) \qquad s^\sharp \wedge \hat{e} : \hat{\tau}_2 \Rightarrow \bot}{s^\sharp \vDash \hat{e} : \hat{\tau}_1}$$

DowncastName

$$\frac{s^\sharp \vDash \hat{e} : \Delta(\hat{\eta})}{s^\sharp \vDash \hat{e} : \hat{\eta}}$$

DowncastWith

$$\frac{s^\sharp \vDash \hat{e}_1 : \hat{\tau} \qquad s^\sharp \vDash \hat{e}_2 \neq 0}{s^\sharp \vDash \hat{e}_1 : (\{x : \hat{\tau} \mid \hat{e}_2\})}$$

DowncastProd

$$\frac{s^\sharp \vDash \hat{e}_1 : \hat{\tau}_1 \qquad s^\sharp \vDash \hat{e}_2 : \hat{\tau}_2}{s^\sharp \vDash (\hat{e}_1 :: \hat{e}_2) : (\hat{\tau}_1 \times \hat{\tau}_2)}$$

DowncastUnion

$$\frac{s^\sharp \vDash \hat{e} : \hat{\tau}_1}{s^\sharp \vDash \hat{e} : (\hat{\tau}_1 \cup \hat{\tau}_2)}$$

DowncastExists

$$\frac{s^\sharp \vDash \hat{e}_1 : \hat{\tau}_1 \qquad s^\sharp \vDash \hat{e}_2 : \mathrm{subst}(\hat{\tau}_2, [\alpha \longrightarrow \hat{e}_1])}{s^\sharp \vDash \hat{e}_2 : (\exists \alpha : \hat{\tau}_1 .\, \hat{\tau}_2)}$$

TypEq

$$\frac{s^\sharp \vDash \hat{e}_1 : \hat{\tau} \qquad s^\sharp \vDash \hat{e}_1 = \hat{e}_2}{s^\sharp \vDash \hat{e}_2 : \hat{\tau}}$$

AllByte

$$\frac{\mathrm{size}(\hat{e}) = \ell}{s^\sharp \vDash \hat{e} : \mathtt{byte}^\ell}$$

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 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 [19, 32] (also called constraint propagation) is an operation which improves the abstract element without changing the concretization.

$$\textsc{Instantiate}\quad \frac{\begin{array}{c} s_0^\sharp \vDash \hat{e} : (\exists \alpha : \hat{\tau}_1 .\, \hat{\tau}_2) \qquad \alpha_1 \text{ fresh} \\ s_0^\sharp \wedge \alpha_1 : \hat{\tau}_1 \Rightarrow s_1^\sharp \qquad s_1^\sharp \wedge \hat{e} : \text{subst}(\hat{\tau}_2, [\alpha \longrightarrow \alpha_1]) \Rightarrow s_2^\sharp \end{array}}{s_0^\sharp \Rightarrow s_2^\sharp}$$

$$\textsc{NumReduc1}\quad \frac{s^\sharp \vDash \hat{e} \neq 0 \qquad s^\sharp.v^\sharp \wedge (\hat{e} \neq 0) \Rightarrow v_1^\sharp}{s^\sharp \Rightarrow s^\sharp[v^\sharp \leftarrow v_1^\sharp]}$$

$$\textsc{NumReduc2}\quad \frac{s^\sharp.v^\sharp \Rightarrow v_1^\sharp}{s^\sharp \Rightarrow s^\sharp[v^\sharp \leftarrow v_1^\sharp]}$$

$$\textsc{TypReduc}\quad \frac{s^\sharp \vDash \hat{e} : \hat{\eta}_1 \star + \hat{e}_1 \qquad s^\sharp \vDash (\hat{\eta}_1, \hat{e}_1) \sqsubseteq_{\mathbb{P}}^\sharp s^\sharp.\Gamma^\sharp[\hat{e}] \vee \hat{e} \notin \text{dom}(s^\sharp.\Gamma^\sharp)}{s^\sharp \Rightarrow s^\sharp[\Gamma^\sharp \leftarrow s^\sharp.\Gamma^\sharp[\hat{e} \leftarrow (\hat{\eta}_1, \hat{e}_1)]]}$$

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 [52] abstract domain), and updates the main domain accordingly.

TypReduc saves typing judgments that have been infered in the abstract type environment $\Gamma^\sharp$. To ensure that we are performing reduction, we only save results that are more precise than what already existed in $\Gamma^\sharp$. The most important thing to note in this rule is that we don't save arbitrary $(\hat{\tau}_1, \hat{e}_1)$ in $\Gamma^\sharp$, only rules of the form $(\hat{\eta}_1, \hat{e}_1)$. The reason is that arbitrary $\hat{e} : \hat{\tau}_1\star + \hat{e}_1$ rules may not be preserved when the store operation performs a mild update, as we discussed in §6; so here we limit the contents of $\Gamma^\sharp$ to only contain typing judgments that are preserved by store. This can be relaxed in two ways; first, because judgments of the form $\hat{e} : (\hat{\tau}_1 \curlywedge \hat{\tau}_2)\star + \hat{e}_1$ will also be preserved when there is a mild update, so they can be saved too. Second, it is possible to save arbitrary judgments $\hat{e} : \hat{\tau}_1\star + \hat{e}_1$ in $\Gamma^\sharp$, provided that the ones that are not preserved are removed when they may be affected by a store operation. This would allow the analysis to be more precise (at the sake of more complexity of the presentation).

D True Alarms and Problems Found while Analyzing the Benchmarks

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

- **Olden/bh** In the function `maketree`, if the mass of all the bodies are not massive, the function `hackcofm` is called with a null pointer, which makes it fail.
- **Olden/mst** The code never checks if pointers returned by `malloc` are null.
- **Shapes/javl** The original code was adapted for use as a benchmark in Li et al. [46], and the macros were changed (notably to remove some `do ... while(0)` constructs). But nested inclusion of these new macros introduced variable capture issues on the `ndir` variable that led to alarms during the analysis of function `jsw_avlinsert`.
- **Shapes/graph** The function `node_add` allocates some memory but does not write the pointer anywhere, leading to a memory leak.

Received 2024-04-06; accepted 2024-08-18