A Language for Probabilistically Oblivious Computation

DAVID DARAIS, University of Maryland
CHANG LIU, University of California, Berkeley
IAN SWEET, University of Maryland
MICHAEL HICKS, University of Maryland

An oblivious computation is one that is free of direct and indirect information leaks, e.g., due to observable differences in timing and memory access patterns. This paper presents $L_{\textbf{obliv}}$, a core language whose type system enforces obliviousness. Prior work on type-enforced oblivious computation has focused on deterministic programs. $L_{\textbf{obliv}}$ is new in its consideration of programs that implement *probabilistic* algorithms, such as those involved in cryptography. $L_{\textbf{obliv}}$ employs a substructural type system and a novel notion of *probability region* to ensure that information is not leaked via the distribution of visible events. The use of regions was motivated by a source of unsoundness that we discovered in the type system of ObliVM, a language for implementing state of the art oblivious algorithms and data structures. We prove that $L_{\textbf{obliv}}$'s type system enforces obliviousness and show that it is nevertheless powerful enough to check state-of-the-art, efficient oblivious data structures, such as stacks and queues, and even tree-based oblivious RAMs.

1 INTRODUCTION

Cloud computing allows clients to conveniently outsource computation, but they must trust that cloud providers do not exploit or mishandle sensitive information. To remove the provider from the trusted computing base, work in both industry and research has strived to realize a "secure" abstract machine comprising an execution engine and protected memory: The adversary cannot see sensitive data as it is being operated on, nor can it observe such data at rest in memory. Such an abstract machine can be realized by encrypting the data in memory and then performing computations using cryptographic mechanisms (e.g., secure multi-party computation [Yao 1986]) or secure processors [Hoekstra 2015; Suh et al. 2003; Thekkath et al. 2000].

Unfortunately, a secure abstract machine does not defend against an adversary that can observe memory access patterns [Islam et al. 2012; Maas et al. 2013; Zhuang et al. 2004] and instruction timing [Brumley and Boneh 2003; Kocher 1996], among other "side" channels of information. For cloud computing, such an adversary is the cloud provider itself, which has physical access to its machines, and so can observe traffic on the memory bus.

A countermeasure against an unscrupulous provider is to augment the secure processor to store code and data in *oblivious RAM* (ORAM) [Maas et al. 2013; Suh et al. 2003]. First proposed by Goldreich and Ostrovsky [Goldreich 1987; Goldreich and Ostrovsky 1996], ORAM obfuscates the mapping between addresses and data, in effect "encrypting" the addresses along with the data. Replacing RAM with ORAM solves (much of) the security problem but incurs a substantial slowdown in practical situations [Liu et al. 2015, 2013; Maas et al. 2013] as reads/writes add overhead that is polylogarithmic in the size of the memory.

Recent work has explored methods for reducing the cost of programming with ORAM. Liu et al. [2015, 2013, 2014] developed a family of type systems to check when *partial* use of ORAM (alongside normal, encrypted RAM) results in no loss of security; i.e., only when the addresses of secret data could (indirectly) reveal sensitive information must the data be stored in ORAM. This optimization can provide order-of-magnitude (and asymptotic) performance improvements.

Working draft as of July 7, 2017.

Wang et al. [2014] explored how to build *oblivious data structures* (ODSs), such as queues or stacks, that are more efficient than their standard counterparts implemented on top of ORAM. Their technique involves specializing ideas from ORAM algorithms to particular data structures, resulting in asymptotic performance gains in common cases. In followup work, Liu et al. [2015] devised a programming language called ObliVM[1] for implementing such oblivious data structures. While Liu et al.'s earlier work treats ORAM as a black box, in ObliVM one can program ORAM algorithms as well as ODSs. A key feature of ObliVM is careful treatment of random numbers, which are at the heart of state-of-the-art ORAM and ODS algorithms. While the goal of ObliVM is that well-typed programs are secure, no formal argument to this effect is made.

In this paper, we present $L_{\mathbf{obliv}}$, a core language for oblivious computation, inspired by ObliVM. $L_{\mathbf{obliv}}$ extends a core imperative language equipped with an information flow type system [Sabelfeld and Myers 2006] with primitives for generating and using uniformly distributed random numbers. We prove that $L_{\mathbf{obliv}}$'s type system guarantees probabilistic *memory trace obliviousness* (MTO), i.e., that the possible distribution of adversary-visible execution traces is independent of the values of secret variables. This property generalizes the deterministic MTO property enforced by Liu et al. [2015, 2013], which did not consider the use of randomness. In carrying out this work, we discovered that the ObliVM type system is unsound, so an important contribution of $L_{\mathbf{obliv}}$ is to address the problem without overly restricting or complicating the language.

$L_{\mathbf{obliv}}$'s type system aims to ensure that no probabilistic correlation forms between secrets and publicly revealed random choices. With oblivious computations it is frequently the case that a random choice is made (e.g., where to store a particular block in an ORAM), and eventually that choice is made visible to the adversary (e.g., when that or another block is later looked up). This visibility is OK as long as the revealed value does not communicate information about a secret, which $L_{\mathbf{obliv}}$ ensures by guaranteeing the number is always (still) uniformly distributed.

$L_{\mathbf{obliv}}$'s type system, presented in Section 4, ensures uniform revelations in part by treating randomly generated numbers as *affine*, meaning they cannot be freely copied. This prohibition prevents revealing the same number twice (where the second revelation will not be uniform). Unfortunately, affinity is too strong for practical algorithms, which do make copies of random numbers. $L_{\mathbf{obliv}}$'s type system thus allows random numbers to be coerced to merely-secret numbers but only so long as the secret version is never revealed. Moreover, $L_{\mathbf{obliv}}$ requires that previously-random numbers not be allowed to influence the choice of to-be-revealed random numbers, since such influence might violate the latter's uniformity. For example, we should not be able to coerce a random number to a secret, look at the secret, and then decide to reveal that random number or some other one. The type system prevents this problem by using a new mechanism we call *probability regions* to track the transitive (in)dependence of numbers in one region from those in another. (That probability regions are missing in ObliVM is the source of its unsoundness.) Section 5 outlines proof that $L_{\mathbf{obliv}}$ enjoys MTO, i.e., if a $L_{\mathbf{obliv}}$ program is well typed, then its observable behavior is oblivious. Further details of the proof are in the Appendix.

While we have not retrofitted $L_{\mathbf{obliv}}$'s type system into ObliVM, we have implemented a type checker for $L_{\mathbf{obliv}}$. Section 6 presents a version of *Tree ORAM* [Shi et al. 2011], a state-of-the-art ORAM implementation, that our language type checks. We also show, in Section 3, that *oblivious stacks*, a kind of oblivious data structure [Wang et al. 2014], can be type checked when implemented on top of a non-recursive ORAM. As far as we are aware, our implementations constitute the first automated proofs that these data structures are indeed oblivious. We believe $L_{\mathbf{obliv}}$ is a promising step forward, generalizing the work that came before it (Section 7 discusses related work).

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

```
let add (mem : (int S * int S) array) (a : int S) (v : int S) =
  let len = length(m) in
  let rec iterate (curr_idx : int P) (pair : (int S * int S)) : unit =
    if curr_idx = len then ()
    else (* WRONG code first *)
      (* let (_, curr_addr) = mem[curr_idx] in *)
      (* if curr_addr = -1 then mem[curr_idx] ← pair *)
      (* else iterate (curr_idx + 1) pair *)
      (* CORRECT code below *)
      let (_, curr_addr) as curr_pair = mem[curr_idx] in
      let (_, addr) = pair in
      let (t_pair, f_pair) = mux (curr_addr = -1 && not(addr = -1)) pair curr_pair in
      mem[curr_idx] ← t_pair;
      iterate (curr_idx + 1) f_pair in
  iterate 0 (v, a)
```

Fig. 1. Adding to an "association array" obliviously (aka add for trivial ORAM)

2 BACKGROUND

This section presents the definition of the threat model and background on type-enforced deterministic oblivious execution. The next section motivates and sketches our novel type system for enforcing probabilistic oblivious execution.

2.1 Threat Model

We assume a powerful adversary that can make fine-grained observations about a program's execution. In particular, we use a generalization of the *program counter (PC) security model* [Molnar et al. 2006]: The adversary knows the program being executed, can observe the PC during execution as well as the contents and patterns of memory accesses. Some *secret* memory contents may be encrypted (while *public* memory is not) but all memory addresses are still visible.

As a relevant instantiation, consider an untrusted cloud provider using a secure processor, like SGX [Hoekstra 2015]. Reads/writes to/from memory can be directly observed, but secret memory is encrypted (using a key kept by the processor). The pattern of accesses, timing information, and other system features (e.g., instruction cache misses) provide information about the PC. Another instantiation is secure multi-party computation (MPC) using secret shares [Goldreich et al. 1987]. Here, two parties simultaneously execute the same program (and thus know the program and program counter), but certain values, once entered by one party or the other, are kept hidden from both using secret sharing.

Our techniques can also handle weaker adversaries, such as those that can observe memory traffic but not the PC, or can make timing measurements but cannot observe the PC or memory.

2.2 Oblivious execution

Our goal is to ensure *memory trace obliviousness (MTO)*, which is a kind of noninterference property [Goguen and Meseguer 1982; Sabelfeld and Myers 2006]. This property states that despite being able to observe each address (of instructions and data) as it is fetched, and each public value, the adversary will not be able to infer anything about manipulated secret values. To see how such an inference could occur, consider the program in Figure 1 (in OCaml-style syntax). This program takes an array of pairs of secret integers mem, a secret address a, and a secret value v and adds

the pair (v,a) to a *free slot* in the array, as defined by its value having address −1. The commented out code on lines 6–8 illustrate a non-oblivious implementation. The code works by (on line 6) extracting the current array element, (on line 7) checking if its address is −1 and updating the element if so, and otherwise (on line 8) continuing to consider the next array element. Notice that curr_index is designated as non-secret (int P means "public int") because the index to an array is effectively revealed when the adversary can observe memory accesses. If the program executes mem[idx] then the adversary can observe some address a being read. Knowing the starting address s of array mem, the adversary can compute idx $= a - s$.

By watching the address trace, an adversary can learn something about the array's contents. In particular, a sequence of N reads followed by a write implies that the first $N - 1$ slots were not free while slot N was; no writes means no free slots. The adversary can also watch the program counter to notice when the true vs. the false branch of the conditional is taken, revealing the same information. The code on lines 10–16 fixes these problems. The **mux** on line 12 is the key part. It takes three arguments. The first is a boolean condition, which here is whether the current slot is free and also whether pair's address is not −1. If so then the **mux** evaluates to a pair comprised of the **mux**'s second and third arguments, in that order; otherwise the pair's contents are reversed. Line 13 updates mem's current slot with the first element of the pair, and recurses to consider the next element, passing the second element of the pair. This code is oblivious: it reads and writes every slot in the array, and always executes exactly the same statements, no matter the contents of the array, index, or value arguments. As such, observing the code's execution reveals nothing about the secrets it manipulates.

This code is actually the add operation of a "trivial" *oblivious RAM* (ORAM) implementation [Goldreich 1987; Goldreich and Ostrovsky 1996]. ORAM provides an API that is like an array, but makes sure the address trace does not reveal the relationship between the index and the value. That is, for read oram i and write oram i v operations, the index i is effectively kept secret. We will later consider sophisticated ORAM algorithms, which use trivial ORAM as a building block.

2.3 Obliviousness by typing

Liu et al. [2015, 2013] developed a type system that ensures programs are MTO. Types are extended to indicate where values are allocated; as per our above example data can be public or secret, but can also reside in ORAM. Liu et al.'s small-step operational semantics $\sigma; e \longrightarrow^{t} \sigma'; e'$ reduces an expression e in memory σ to heap σ' and expression e' while emitting trace event t. Trace events include fetched instruction addresses, public values, and the addresses of public and secret values, that are read and written, but not the addresses or values of data stored in ORAM, which is modeled as a black box: Basically it is treated as if it were an array whose accesses are opaque to the adversary (similar to the Dolev-Yao modeling of encrypted messages [Dolev and Yao 1981]). Under this model, the MTO property means that running e in two *low-equivalent* memories σ_1 and σ_2—meaning they agree on public values—will produce the exact same memory trace, along with low-equivalent output heaps and results. More formally, if $\sigma_1 \sim \sigma_2$ then $\sigma_1; e \longrightarrow^{t_1} \sigma_1'; e_1$ and $\sigma_2; e \longrightarrow^{t_2} \sigma_2'; e_2$ imply $t_1 = t_2$, $\sigma_1' \sim \sigma_2'$, and $e_1 \sim e_2$, where operator $\sim$ denotes low-equivalence.

3 TYPE-BASED ENFORCEMENT OF PROBABILISTIC OBLIVIOUSNESS

Liu et al's type system enforces obliviousness of *deterministic* programs that use ORAM. However, work on *oblivious data structures* [Wang et al. 2014] has shown that *probabilistic* uses of ORAM can be far more efficient. In such uses, visible memory and traces need not be identical when computing with different secrets, but rather they only must be *identically distributed*; i.e., all possible memory traces are equally likely, no matter the secrets used. Indeed, this is the same property that ORAM

```
1  let stackop ((oram,rid_r) :' a Stack) (ispush:Bool S) (d :' a): 'a =
2  let rid = ! rid_r in
3  let old_d = read oram rid in
4  let (d ', _) = mux ispush d old_d in
5  let (id ,_) = mux ispush (rid+1) rid in
6  write oram id d ';
7  let (rid ', _) = mux ispush (rid+1) (rid-1) in
8  rid_r := rid '; d'
```

Fig. 2. A deterministic oblivious stack built using a full (recursive) ORAM.

algorithms, if we open up the black box, are actually providing. This section motivates the benefits of probabilistic programming with ORAMs, and sketches our type system that enforces MTO for such programs. Section 6 shows this type system is actually powerful enough to implement ORAM algorithms directly, so they need not be treated as black boxes.

3.1 Motivation: naive oblivious stack is inefficient

The goal of an *oblivious stack* is to hide both its data and which operations (pushes or pops) are taking place—only the total number of operations should be revealed. An incorrect approach would be to implement the stack as usual but using a secret (encrypted) array. While doing this would hide the contents of the array, indexes into that array would be visible to the adversary (as discussed in Section 2.2), which would reveal the operation—increasing indexes indicate pushes, and decreasing ones indicate pops. To hide the index we can store the stack in an ORAM instead of an array, and merge the code of the push and pop operations, as shown in the `stackop` function in Figure 2, so that observation of the PC does not reveal which operation is taking place.

In the code, a stack consists of an ORAM of secret numbers and a reference storing the index of the root. Function `stackop` takes a stack, a flag indicating whether the operation is a push or pop, and the value to push, and returns a value. The code reads the value at the root index (line 3). The next line copies that value to `d'` if the operation is pop, or else puts `d` there if it is a push. Line 5 determines the index of the write it will perform on line 6: this index (`id`) is one more than the root index if it's a push and it's the current root index if not. As such, the write on line 6 puts the given value in the next slot in case of a push, or writes back the value at the current root, if it's a pop. Finally, line 7 adjusts the root index, and line 8 returns the result, which is either the popped value or pushed value (if it was a push).

While this code works perfectly well (and would typecheck in Liu et al's system) a *probabilistic* version of the stack, using a *non-recursive ORAM*, would be more space-efficient. In particular, it will require only $O(L)$ extra space where L is the current size of the stack, whereas this version requires $O(N)$ extra space, where N is the size of the ORAM. We will see why, next.

3.2 Recursive and non-recursive ORAM

To understand the source of inefficiency, we now digress to consider features of state-of-the-art ORAM algorithms. In particular, consider a tree-based ORAM [Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015]. It has two parts: a tree-like structure for storing the actual data blocks, and a *position map* that maps logical data block indexes to *position tags* that indicate the block's position in the tree. In the simplest instantiation, an ORAM of size N has a position map of size N which is hidden from the adversary; e.g., it could be stored in on-chip memory in a processor-based deployment of ORAM. The position tags mask the relationship between a logical index and the

location of its corresponding block in the tree. As blocks are read and written, they are shuffled around in the data structure, and their new locations are recorded in the position map. Note that the position map need not be hidden on chip; rather, much of it can be stored recursively in ORAM itself, reducing the space overhead on the client. As such, the tree part that contains the data blocks is sometimes called a *non-recursive ORAM* (or NORAM for short).

The ORAM read and write operations are implemented as a sequence of two NORAM operations, called readAndRemove (or rr, for short) and add.

```
rr :  'a NOram → (idx:int S) → (tag:int P) → 'a
add:  'a NOram → (idx:int S) → (tag:int S) → (block:'a) → unit
```

Here, the type 'a NORam designates the tree portion of the ORAM data structure, which stores data blocks of type 'a. Operation rr's int S argument is the desired block's logical index, which is hidden from the adversary; the int P argument is its position tag extracted from the map; and the 'a is the returned data block. The add operation is similar except that its position tag can be secret, and it adds a block to the tree, rather than removing it. A full ORAM read at index i requires looking up the tag in the position map, performing an rr to read and remove the value at i, and adding back that value (updating the position map with a fresh tag). A write to i requires a rr to remove the old value at i, and an add to write the new value (again updating the position map).

Depending on the details of the NORAM algorithm, rr and add may perform many reads and writes to/from the tree. For simplicity, we can think of a call to rr noram id pos as producing a single abstract event RR(noram,pos). The event indicates the operation, the NORAM it is performed on (there may be more than one), and the position tag, which is visible because it correlates with the actual memory addresses accessed to retrieve the requested block. The add operation's behavior does not depend on the position tag, so it simply generates event ADD(noram). The fact that rr's position tag is visible to the adversary means that traces involving rr operations might differ not because of a dependence on particular secret values, but due to innocuous random variation. Hence we require a more general definition of obliviousness, discussed below.

The running time of both rr and add are $O(\log N)$. The position map requires $O(N)$ storage and $O(1)$ access time when stored non-recursively on the client, hidden from the adversary. When stored recursively in ORAM, the position map imposes $O(1)$ storage on the client, and $O(\log N$ access time; this is the standard deployment for (full, recursive) ORAM, and is necessary when the position map accesses are visible to the adversary (as in MPC-based deployments).

It turns out that $L_{\textbf{obliv}}$ is powerful enough to implement tree-based ORAMs from scratch; we present more details about this in Section 6.

3.3 An optimized oblivious stack

Returning to the question of how to more efficiently implement an oblivious stack, consider that we always access a stack via its head, using the root index. Thus, in the code in Figure 2, the full `oram` internally only ever uses one slot in its position map. Thus we can do better by using an NORAM directly, having the stack manage the position tag of the root. In short, we implement an oblivious stack as a triple comprising a NORAM, the index of the root element, and its position tag. The latter two act as a kind of pointer into the NORAM. Each block stored in the NORAM contains the data and the position tag of the next block in the stack.

Code implementing the stack following this design is given in Figure 3. Note the code branches on the ispush variable to make it easier to read; the actual implementation must use `mux`s to conditionally execute each statement in both branches to ensure obliviousness.[2] Line 3 extracts the current root

[2]Notice that the structure of both branches is roughly parallel, which makes converting to the use of **mux**es straightforward.

```
1  type 'a OStack = ('a * rnd R) noram * int S ref * rnd R ref
2  let stackop ((noram,rid_r,pos_r):'a OStack) (ispush:Bool S) (d:'a) : 'a =
3  let (rid,pos) = !rid_r,!pos_r in
4  let (rid',pos',d') =
5    if ispush then
6      let (d',_) = rr noram (-1) (reveal (rnd())) in
7      let b = (d,pos) in
8      let pos' = rnd() in
9      let _ = add noram (rid+1) (use pos') b in
10     (rid+1,pos',d')
11   else
12     let (d',pos') = rr noram rid (reveal pos) in
13     let b = (d,rnd()) in
14     let _ = add noram (-1) 0 b in
15     (rid-1,pos',d') in
16 rid_r := rid';
17 rpos_r := pos';
18 d'
```

Fig. 3. A probabilistic oblivious stack built using a non-recursive ORAM. (Does not use **mux**, for simplicity.)

index and position tag. Lines 6–10 handle a push operation. Line 6 first does a "dummy read" from the NORAM; just as we saw with the oblivious `add` in Figure 1, using index -1 results in a dummy block being returned (the position tag argument is unimportant in this case). Line 7 constructs a new block `b` to push: it consists of the given data `d` paired with the current root's position tag `pos`, thus creating a "pointer" to that block. We then generate a fresh position tag `pos'` for this (the new root's) block, add the block to the `noram`. Random numbers generated by `rnd()` have type `rnd R`; the coercion **reveal** ascribes a random number the type `int P` (per line 6), while **use** gives it type `int S` (line 9); we explain these constructs in the next subsection. The new root index (the old one plus one), the root's tag, and the dummy block passed in are returned on line 10. Lines 12–15 handle a pop. Here, the first `rr` does real work, extracting the block that corresponds to the root index and position tag. We then generate a dummy block to "add" to the ORAM. The updated root index (the old one minus one), its position tag (returned by the `rr`) and the fetched block are returned.

This version of an oblivious stack performs better than the version from Figure 2. The space overhead, due to the added pointers at the root and within the ORAM, is $O(L)$ where L is the size of the current stack, not the size N of the ORAM. The running time is still $O(\log N)$.

The random numbers generated on lines 6 and 8 are eventually revealed by `rr` operations on lines 4 and/or 12. As such, there can be many possible $RR(\ldots)$ events for the same sequence of secret inputs. While this code is safe, there is the possibility that a difference in the visible trace reveals secret information. We need to ensure that traces are *identically distributed*. More formally, if $\sigma_1 \sim \sigma_2$ then for all traces t the *probability* that evaluating e under σ_1 produces trace t is the same as the probability that evaluating e under σ_2 also produces t. This property, which we call *probabilistic trace obliviousness*, is ensured by the underlying NORAM implementation, and we need to ensure that the programs that use it preserve it. It turns out this property holds for our stack, and is confirmed by our type system, as described next.

3.4 $L_{\textbf{obliv}}$: Obliviousness by typing for probabilistic programs

The main contribution of this paper is $L_{\textbf{obliv}}$, a core language for ensuring obliviousness of programs that use randomness. $L_{\textbf{obliv}}$'s type system establishes that programs like the one in Figure 3 are oblivious by employing two mechanisms: *affine* treatment of random values, and *probability regions* to track dependences between random values that could leak information when a value is revealed.

Affinity. Random numbers, produced by executing `rnd()`, are given type `rnd R`, where `R` is a probability region, explained shortly. Values of this type are, like secret values, invisible to the adversary. They can be converted to `int S` numbers with the **use** coercion and can be converted to `int P` numbers with the **reveal** coercion. For example, in Figure 3, line 8 creates a random value `pos'`, which it converts to a secret number and passes to the `add` operation on line 9, and sets to be the position tag of the root on line 10. On line 12, the root position tag is converted to a public number which is passed as an argument to `rr` (effectively revealing it to the adversary).

$L_{\textbf{obliv}}$'s type system enforces that a random number can be made public at most once. It does this by treating values of type `Rnd R` affinely: they cannot be duplicated, e.g., through assignments, and they are consumed by operations other than **use**. For example, variable `pos'` is not consumed on line 9 when passed to **use**, but is consumed when assigned to `rpos_r` on line 17; subsequent reference to `n_rpos` on line 18 would be illegal. Similarly, on line 3 variable `pos` is read from the root of the tree; it has type `Rnd R`. It is then consumed on line 6 when given to **reveal**.

Why is affinity important? Consider this example, in which variable `s` has type `int S`.

```
1  let (r1,r2) = (rnd(), rnd()) in
2  let (z,_) = mux (s = 0) r1 r2 in
3  output (reveal z); (* OK *)
4  output (reveal r1); (* Bad *)
```

Lines 1–3 in this code are safe: we generate two random numbers that are invisible to the adversary, and then store one of them in `z` depending on whether the secret `s` is zero or not. Revealing `z` at line 3 is safe: regardless of whether `z` contains the contents of `r1` or `r2`, the fact that both are uniformly distributed means that whatever is revealed, nothing can be learned about `s`. However, revealing `r1` on line 4, after having revealed `z`, is not safe. This is because seeing two zeroes in a row is more likely when `z` is `r1`, which happens when `s` is zero. So this violates obliviousness.

A similar problem would arise in our oblivious stack if we changed line 8 in Figure 3 to be `let pos' = pos in ...` This would set the position tag for the new root to the old root's tag, `pos`, which has just been revealed. As a result, we will leak information about the secret `ispush` variable at the next operation, i.e., whether a push was previously performed, vs. a pop. With affine revelation, the type system catches this mistake; assigning `pos` to `b` on line 7 consumes it, so reading it again on line 8 would be disallowed.

Probability regions. Unfortunately, affinity on its own is insufficient. This is because, for reasons of needed expressiveness, **use** can convert random numbers to non-affine, secret ones, and then computing on these can affect the distribution of revealed values. To see how, consider this program:

```
1  let (x,y) = (rnd (), rnd ()) in
2  let s_x = use(x) in
3  let (r1,r2) = mux (s_x = 0) x y in
4  let (z,_) = mux (s = 0) r1 r2 in
5  output (reveal z); (* Bad *)
```

Line 1 generates two random numbers, and line 2 creates a secret copy of `x`, *which does not consume it.* Line 3 sets `r1` to `x` if it is 0, and to `y` otherwise; the reverse is true for `r2`. The problem now is that `r1` and `r2` are not uniformly distributed: `r1` is more likely to contain zero than any other number. As

a result, outputting z on line 5 is dangerous: seeing a zero here is more likely if z was r1 and thus that says that s is more likely to be zero, than not. Notice that we have not violated affinity here: no random number has been revealed publicly more than once.

Our type system addresses this problem with a novel construct we call *probability regions*. Both normal and random number types are ascribed a region R. The region represents the set of random numbers generated at program points whose rnd instructions are annotated with R. (We have elided the region name so far, but normally should write rnd R ()). Regions allow the type system to reason that random numbers are probabilistically independent. In particular, regions follow a lattice ordering, and we can say that two regions R1 and R2 are independent if their meet is ⊥. Otherwise, regions may depend on one another, meaning that revelation of a random number in one region may compromise the uniformity of the distribution of a number in the other, which could result in a leak.

We can see this in our example: the region of the number s_x = 0 is the same region as x, since s_x was derived from x. As such, the two are not independent. But this lack of independence is problematic when conditioning on s_x = 0 to return x; it means that the output is no longer uniform. On the other hand, if the guard of a **mux** has a region that is *independent* of the regions of its branches, then the uniformity of the output is not threatened. Normal numbers that never were (or were influenced by) random numbers have region ⊥ because they are independent of any random number; as a notational convenience, when we write only the secrecy label (e.g., int P or int S) this means the region is ⊥. Looking at our first example, line 2 is perfectly safe assuming that s has type int S because then its region ⊥ is independent of the region of r1 and r2. Allowing such safe code is critical for implementing efficient oblivious algorithms.

4 FORMAL LANGUAGE

This section presents $L_{\mathbf{obliv}}$: Its syntax, sampling semantics, and type system. The next section presents our main metatheoretic result, that typing ensures memory trace obliviousness.

4.1 Syntax

Figure 4 shows the syntax for $L_{\mathbf{obliv}}$. The syntax for terms is stratified into a kind of *a-normal form* to simplify the semantics. We have expressions e, which comprise let binding and conditionals; atomic expressions a, which comprise various computational forms discussed shortly; and pico expressions p, which comprise variables x or literals tagged with a security label ℓ. Literals are either bits b (which are either `O` or `I`) or natural numbers n, and security labels ℓ are either public `P` or secret `S`. Semantically, literals labeled with `S` are invisible to the adversary while those labeled with `P` are not. (Think of `S`-labeled literals as encrypted.)

Returning to atomic expressions, p° takes the complement of a bit; $p \odot p$ represents an arbitrary arithmetic operation; $\texttt{asnat}(p)$ converts from bits to nats, and $\texttt{asbit}(p)$ does the reverse. Expression $\texttt{flip}^\rho()$ randomly generates a bit of type $\texttt{flip}^\rho$; we call values of this type "coin flips." The annotation ρ is the coin flip's probability region; we defer discussion of these to Section 4.3. Coin flips are semantically secret, and have limited use; we can coerce one to public bit via $\texttt{reveal}(p)$ and to a secret bit via $\texttt{use}(p)$. Expression $\texttt{mux}(p, p_2, p_3)$ unconditionally evaluates p_2 and p_3 and returns them as a pair in the given order if p evaluates to `I`, or in the opposite order if it evaluates to `O`. This operation is critical for obliviousness because its operation is atomic. By contrast, normal conditionals $\texttt{if } p \texttt{ then } e_1 \texttt{ else } e_2$ evaluate either e_1 or e_2 depending on p, never both, so the instruction trace communicates which branch is taken. The components of tuples constructed as $\langle p_1, p_2 \rangle$ can be accessed via $\texttt{let } x_1, x_2 := a \texttt{ in } e$. Finally, the last three atomic expression forms are for (constant size-N) array creation, read, and update, respectively.

$\ell \in$	*label*	$::= \texttt{P} \mid \texttt{S}$	security label
$\rho \in$	*region*	$::=$ *(parameter)*	probability region
$b \in$	$\mathbb{B}$	$::= \texttt{O} \mid \texttt{I}$	bits
$n \in$	$\mathbb{N}$	$::= \{0, 1, \ldots\}$	natural numbers
$\odot \in$	*aop*	$::=$ *(parameter)*	arithmetic ops
$x \in$	*var*	$::= \{\texttt{x}, \texttt{y}, \texttt{z}, \ldots\}$	lexical variables
$\iota \in$	*lit*	$::= b \mid n$	literals
$p \in$	*pico*	$::= x \mid \iota_\ell$	"pico" expressions
$a \in$	*atom*	$::= p$	"atomic" expressions
		$\mid p^\circ$	bit complement
		$\mid p \odot p$	arithmetic ops
		$\mid \texttt{asnat}(p)$	bit-to-nat conversion
		$\mid \texttt{asbit}(p)$	nat-to-bit conversion
		$\mid \texttt{flip}^\rho()$	coin flip
		$\mid \texttt{use}(x)$	convert flip to bit
		$\mid \texttt{reveal}(x)$	reveal flip to public
		$\mid \texttt{mux}(p, p, p)$	atomic conditional
		$\mid \langle p, p \rangle$	tuple creation
		$\mid \texttt{array}^N(p, \ldots_N, p)$	array creation
		$\mid p[p]$	array read
		$\mid p[p] \leftarrow p$	atomic array read+write
$e \in$	*expr*	$::= a$	"compound" expressions
		$\mid \texttt{let } x \coloneqq a \texttt{ in } e$	local variable binding
		$\mid \texttt{let } x, x \coloneqq a \texttt{ in } e$	local tuple binding
		$\mid \texttt{if } p \texttt{ then } e \texttt{ else } e$	conditional

Fig. 4. Syntax for $L_{\textbf{obliv}}$

We designed $L_{\textbf{obliv}}$ to capture the key features used when programming oblivious data structures (notably, numbers, tuples, arrays, and random numbers), but it omits obviously useful features to avoid unimportant complexity. Many missing features can be added without difficulty (and we have them in our implementation). For example, we can easily encode random N-bit numbers as a tuple of bits, with ready conversions to/from nats, and we can allocate variable-sized arrays. We can support lambda terms with bounded recursion using standard ideas; we imagine $L_{\textbf{obliv}}$ programs as post inlining. Allowing non-termination (as opposed to bounded recursion) is more technically delicate, but also doable. We further discuss implementation issues (including the support for richer types) in Section 6.

4.2 Sampling Semantics

Figure 5 shows a small-step operational semantics for $L_{\textbf{obliv}}$ programs. The main judgment has form $\varsigma \rightsquigarrow^q \varsigma'$ which states that a configuration ς steps to configuration ς' with probability q (a rational number). A configuration contains an expression e, environment γ, and a store σ; a special HALT form indicates a terminated computation, with the expression replaced with a value v. A

value is a labeled literal ι_ℓ, an address, or a pair of values $\langle v, v \rangle$. The environment maps variables to values, while the store maps (array) addresses α to sequences of values.

The bottom of the figure gives rules for the main judgment, and are basically standard; notice that let-bound variables are put in the environment γ, rather substituted into the target term. These rules inherit their probability annotation from subsidiary judgment $\gamma \ ; \ \sigma \vdash a \twoheadrightarrow^q v \ ; \ \sigma'$, which states that an atomic expression a under γ and σ steps to a value v and output store σ' with probability q. In turn, these rules invoke $\mathcal{P}[\![p]\!]$ to evaluate pico expressions—variables x are looked up in the environment, and literals ι_ℓ are returned. All of the atomic expression rules have probability $q = 1$ except for the rules for `flip`, which return $\mathtt{O}_\mathsf{S}$ or $\mathtt{I}_\mathsf{S}$ with probability $1/2$. The rules preceding these are for computing on bits and numbers, and are straightforward; we assume a given semantics function for operations $\odot$. The rules for `mux` evaluate both branches and their values as a pair; the order of these values depends on the bit to which the guard evaluates. Array allocation stores a sequence of the given values at fresh address α in the store; array read extracts a value at the given index n_P (notice it must be public), and array write returns the existing value before updating the appropriate position in the sequence present at α in the store. Having array update return the old value is critical for coexisting with the type system's affine treatment of random numbers, as described below. Attempting to access an array out of bounds is undefined; note that this does not represent an information leak because array indexes are public.

Traces and adversary observations. An execution trace t is a sequence of configurations $\varsigma_1 \ldots \varsigma_N$ whereby each ς_i and ς_{i+1} in the sequence is justified by proof of judgment $\varsigma_i \rightsquigarrow^q_{i+1}$, and ς_N is a halted configuration. We define what parts of the trace the adversary can see according to the function *obs*. This function takes a sequence of configurations and produces a corresponding sequence of "bulleted" configurations. We omit this function for space reasons (it is given in the appendix, in Figure 21) but its operation is straightforward: a bullet configuration is the same as its original except that all secret numbers (both random and non-random) have been replaced with $\bullet$. As such, we capture the idea that the adversary can see the full instruction trace (the structure of expressions is visible) the address trace (addresses α are not affected), and non-secret values (public literals are unaffected). Only specifically secret numbers are hidden. Our type system, given next, is able to ensure even when the adversary can see such a trace, he can learn nothing about the program's secret values.

4.3 Typing

Figure 7 shows the type system for $L_{\mathbf{obliv}}$, which defines a judgment $\Gamma \vdash e : \tau \ ; \ \Gamma'$, which states that under type environment Γ expression e has type τ, and yields residual type environment Γ'. Types τ and environments Γ are defined in Figure 6. Bits and numbers respectively have type $\mathtt{bit}^\rho_\ell$ and $\mathtt{nat}^\rho_\ell$, where ℓ and ρ are the label and probability region, respectively. As mentioned earlier, $\mathtt{flip}^\rho$ characterizes a coin flip in region ρ; it implicitly has label S. Types $\mathtt{array}(\tau)$ and $\tau \times \tau$ characterize arrays and pairs, respectively. Environments map variables to either types τ or inaccessibility tags $\bullet$; the latter is used to support affinity, discussed shortly.

At a high level, the type system is enforcing three properties:

(1) No type errors occur at runtime (e.g., operating on an array as if it were a number).
(2) No secrets can be inferred from adversary-visible information.
(3) No probabilistic correlation forms between secrets and publicly revealed random numbers.

Properties (1) and (2) are enforced using standard techniques. For example, no type-correct program can operate on an array as if it were a number because rule Aop only types expression $p_1 \odot p_2$ when p_1 and p_2 are numbers (not arrays). Moreover, this operation correctly captures the influence of

$$\alpha \in \mathit{addr} ::= \mathbb{N} \qquad \upsilon \in \mathcal{V} ::= \iota_\ell \mid \alpha \mid \langle \upsilon, \upsilon \rangle$$

$$\gamma \in \mathit{env} ::= \mathit{var} \rightharpoonup \mathcal{V} \qquad \sigma \in \mathit{store} ::= \mathit{addr} \rightharpoonup \mathcal{V}^* \qquad \varsigma \in \mathit{config} ::= \langle e, \gamma, \sigma \rangle \mid \mathtt{HALT}(\upsilon, \gamma, \sigma)$$

$$\mathcal{P}[\![\cdot]\!] : \mathit{pico} \to \mathit{env} \rightharpoonup \mathit{val} \qquad \mathcal{P}[\![x]\!](\gamma) ::= \gamma(x) \qquad \mathcal{P}[\![x]\!](\iota_\ell) ::= \iota_\ell$$

$$\boxed{\gamma \; ; \sigma \vdash a \twoheadrightarrow^q \upsilon \; ; \sigma}$$

$$\begin{array}{rlll}
\textsc{CompO} & \gamma \; ; \sigma \vdash p^o & \twoheadrightarrow^1 & \mathtt{I}_\ell \; ; \sigma \quad \mathit{when} \; \mathtt{O}_\ell = \mathcal{P}[\![p]\!](\gamma) \\
\textsc{CompI} & \gamma \; ; \sigma \vdash p^o & \twoheadrightarrow^1 & \mathtt{O}_\ell \; ; \sigma \quad \mathit{when} \; \mathtt{I}_\ell = \mathcal{P}[\![p]\!](\gamma) \\
\textsc{Aop} & \gamma \; ; \sigma \vdash p_1 \odot p_2 & \twoheadrightarrow^1 & [\![\odot]\!](\mathcal{P}[\![p_1]\!](\gamma), \mathcal{P}[\![p_2]\!](\gamma)) \; ; \sigma \\
\textsc{AsNO} & \gamma \; ; \sigma \vdash \mathtt{asnat}(p) & \twoheadrightarrow^1 & 0_\ell \; ; \sigma \quad \mathit{when} \; \mathtt{O}_\ell = \mathcal{P}[\![p]\!](\gamma) \\
\textsc{AsNI} & \gamma \; ; \sigma \vdash \mathtt{asnat}(p) & \twoheadrightarrow^1 & 1_\ell \; ; \sigma \quad \mathit{when} \; \mathtt{I}_\ell = \mathcal{P}[\![p]\!](\gamma) \\
\textsc{AsB} & \gamma \; ; \sigma \vdash \mathtt{asbit}(p) & \twoheadrightarrow^1 & (n \,\%\, 2)_\ell \; ; \sigma \quad \mathit{when} \; n_\ell = \mathcal{P}[\![p]\!](\gamma) \\
\textsc{FlipO} & \gamma \; ; \sigma \vdash \mathtt{flip}^\rho & \twoheadrightarrow^{1/2} & \mathtt{O}_\mathsf{S} \; ; \sigma \\
\textsc{FlipI} & \gamma \; ; \sigma \vdash \mathtt{flip}^\rho & \twoheadrightarrow^{1/2} & \mathtt{I}_\mathsf{S} \; ; \sigma \\
\textsc{Use} & \gamma \; ; \sigma \vdash \mathtt{use}(x) & \twoheadrightarrow^1 & \gamma(x) \; ; \sigma \\
\textsc{Rev} & \gamma \; ; \sigma \vdash \mathtt{reveal}(x) & \twoheadrightarrow^1 & \gamma(x) \; ; \sigma \\
\textsc{MuxI} & \gamma \; ; \sigma \vdash \mathtt{mux}(p_1, p_2, p_3) & \twoheadrightarrow^1 & \langle \mathcal{P}[\![p_2]\!](\gamma), \mathcal{P}[\![p_3]\!](\gamma) \rangle \; ; \sigma \quad \mathit{when} \; \mathtt{I}_\ell = \mathcal{P}[\![p_1]\!](\gamma) \\
\textsc{MuxO} & \gamma \; ; \sigma \vdash \mathtt{mux}(p_1, p_2, p_3) & \twoheadrightarrow^1 & \langle \mathcal{P}[\![p_3]\!](\gamma), \mathcal{P}[\![p_2]\!](\gamma) \rangle \; ; \sigma \quad \mathit{when} \; \mathtt{O}_\ell = \mathcal{P}[\![p_1]\!](\gamma) \\
\textsc{Tup} & \gamma \; ; \sigma \vdash \langle p_1, p_2 \rangle & \twoheadrightarrow^1 & \langle \mathcal{P}[\![p_1]\!](\gamma), \mathcal{P}[\![p_2]\!](\gamma) \rangle \; ; \sigma \\
\textsc{Arr} & \gamma \; ; \sigma \vdash \mathtt{array}^N(p_1, \ldots, p_N) & \twoheadrightarrow^1 & \alpha \; ; \sigma[\alpha \mapsto \mathcal{P}[\![p_1]\!](\gamma) \ldots \mathcal{P}[\![p_N]\!](\gamma)] \\
 & & & \quad \mathit{when} \; \alpha = \mathit{fresh}(\sigma) \\
\textsc{Read} & \gamma \; ; \sigma \vdash p_1[p_2] & \twoheadrightarrow^1 & \sigma(\alpha)[n] \; ; \sigma \quad \mathit{when} \; \alpha = \mathcal{P}[\![p_1]\!](\gamma), n_\mathsf{P} = \mathcal{P}[\![p_2]\!](\gamma) \\
\textsc{Write} & \gamma \; ; \sigma \vdash p_1[p_2] \leftarrow p_3 & \twoheadrightarrow^1 & \sigma(\alpha)[n] \; ; \sigma[\alpha \mapsto \sigma(\alpha)[n \mapsto \mathcal{P}[\![p_3]\!](\gamma)]] \\
 & & & \quad \mathit{when} \; \alpha = \mathcal{P}[\![p_1]\!](\gamma), n_\mathsf{P} = \mathcal{P}[\![p_2]\!](\gamma)
\end{array}$$

$$\boxed{\varsigma \rightsquigarrow^q \varsigma}$$

$$\textsc{Atom}\frac{\gamma \; ; \sigma \vdash a \twoheadrightarrow^q \upsilon \; ; \sigma'}{\langle a, \gamma, \sigma \rangle \rightsquigarrow^q \mathtt{HALT}(\upsilon, \gamma, \sigma')} \qquad \textsc{Let}\frac{\gamma \; ; \sigma \vdash a \twoheadrightarrow^q \upsilon \; ; \sigma'}{\langle \mathtt{let} \; x := a \; \mathtt{in} \; e, \gamma, \sigma \rangle \rightsquigarrow^q \langle e, \gamma[x \mapsto \upsilon], \sigma' \rangle}$$

$$\textsc{LetT}\frac{\gamma \; ; \sigma \vdash a \twoheadrightarrow^q \langle \upsilon_1, \upsilon_2 \rangle \; ; \sigma'}{\langle \mathtt{let} \; x_1, x_2 := a \; \mathtt{in} \; e, \gamma, \sigma \rangle \rightsquigarrow^q \langle e, \gamma[x_1 \mapsto \upsilon_1, x_2 \mapsto \upsilon_2], \sigma' \rangle}$$

$$\textsc{IfI}\frac{\gamma \; ; \sigma \vdash a \twoheadrightarrow^q \mathtt{I}_\mathsf{P} \; ; \sigma'}{\langle \mathtt{if} \; p \; \mathtt{then} \; e_1 \; \mathtt{else} \; e_2, \gamma, \sigma \rangle \rightsquigarrow^q \langle e_1, \gamma, \sigma' \rangle}$$

$$\textsc{IfO}\frac{\gamma \; ; \sigma \vdash a \twoheadrightarrow^q \mathtt{O}_\mathsf{P} \; ; \sigma'}{\langle \mathtt{if} \; p \; \mathtt{then} \; e_1 \; \mathtt{else} \; e_2, \gamma, \sigma \rangle \rightsquigarrow^q \langle e_2, \gamma, \sigma' \rangle}$$

Fig. 5. Sampling Semantics for $L_{\mathbf{obliv}}$

$$\tau \in \mathit{type} ::= \mathtt{bit}^{\rho}_{\ell} \mid \mathtt{nat}^{\rho}_{\ell} \mid \mathtt{flip}^{\rho} \mid \mathtt{array}(\tau) \mid \tau \times \tau \qquad \dot{\tau} \in \dot{\mathit{type}} ::= \tau \mid \bullet \qquad \dot{\tau} \sqsubset \bullet$$

$$\Gamma \in \mathit{tenv} := \mathit{var} \rightharpoonup \dot{\mathit{type}} \qquad \kappa \in \mathit{kind} ::= \mathtt{U} \mid \mathtt{A} \qquad \mathtt{U} \sqsubset \mathtt{A}$$

$$\mathcal{K} : \mathit{type} \to \mathit{kind} \qquad \mathcal{K}(\tau_1 \times \tau_2) := \mathcal{K}(\tau_1) \sqcup \mathcal{K}(\tau_2)$$

$$\mathcal{K}(\mathtt{bit}^{\rho}_{\ell}) := \mathtt{U} \qquad \mathcal{K}(\mathtt{nat}^{\rho}_{\ell}) := \mathtt{U} \qquad \mathcal{K}(\mathtt{flip}^{\rho}) := \mathtt{A} \qquad \mathcal{K}(\mathtt{array}(\tau)) := \mathtt{U}$$

Fig. 6. Type and Kind Language for $L_{\mathbf{obliv}}$

secret information by annotating the result type's security label with $\ell_1 \sqcup \ell_2$, the join of the labels of p_1 and p_2. (Recall that $\mathtt{S} \sqcup \ell = \mathtt{S}$ for all ℓ.) Likewise, the rules ensure that array indexes and conditionals do not leak information improperly. Property (3) is the main novelty of this work, and its need arises from $L_{\mathbf{obliv}}$'s support for random numbers that are used to enforce secrecy but can be later be seen by the adversary.

A key invariant is that coin flips, which have type $\mathtt{flip}^{\rho}$, always have the following properties:

(1) Their distribution is independent of other flip values.
(2) Their distribution is *stable*, meaning the probability of each possible bit value $\mathtt{0}$ or $\mathtt{I}$ is $1/2$.

The type system only allows creating, manipulating, and eliminating values at flip type in ways that preserve properties (1) and (2). In particular, property (1) is maintained by treating flip values as *affine*, which prevents their duplication, and property (2) is maintained by tracking *probability regions* of flip values as they are combined and used in computations.

Affinity. To enforce non-duplicability, when an affine variable is used by the program, its type is removed from the output environment. Figure 6 defines kinding metafunction $\mathcal{K}$ that assigns a type either the kind $\mathtt{U}$ (freely duplicatable) or $\mathtt{A}$ (non-duplicatable). Arrays, nats, and bits are always universal, and flips are always affine; a pair is considered affine if either of its components is. Rule VarU in Figure 7 types universally-kinded variables; the output environment Γ is the same as the input environment. Rule VarA types an affine variable by marking it $\bullet$ in the output environment. This rule is sufficient to rule out the first problematic example in Section 3.4.

There are a few other rules that are affected by affinity. Rules Use and Reveal permit converting $\mathtt{flip}^{\rho}$ types to bits via the `use` and `reveal` coercions, respectively. The first converts a $\mathtt{flip}^{\rho}$ to a $\mathtt{bit}^{\rho}_{\mathtt{S}}$ and does *not* make x inaccessible, while the second converts to a $\mathtt{bit}^{\rho}_{\mathtt{P}}$ and does make it inaccessible. In essence, the type system is enforcing that any random number is made adversary-visible at most once; secret copies (that are never themselves revealed) are allowed.

Arrays are permitted to contain affine or universal values. Only in the latter case is it safe to read out an array element directly, since in the former it would become immediately inaccessible. Hence rule Read requires that array element type τ have kind $\mathtt{U}$. Rule Write applies to all arrays: the returned value is the old one (just as with a read), and the written one serves as its immediate replacement, guaranteeing the store remains well-typed.

Finally, note that different variables could be made inaccessible in different branches of a conditional, so If types each branch in the same initial context, but then joins their the output contents; if a variable is made inaccessible by one branch, it will be inaccessible in the joined environment.

Probability regions. Now we consider the role of probability regions ρ in the type system. Probability regions are a partially ordered set. The purpose of regions is to track fine-grained, dynamic

$$\boxed{\Gamma \vdash e : \tau \;;\; \Gamma}$$

$$\textsc{VarU}\ \frac{\mathcal{K}(\Gamma(x)) = \mathsf{U}}{\Gamma \vdash x : \Gamma(x) \;;\; \Gamma} \qquad \textsc{VarA}\ \frac{\mathcal{K}(\Gamma(x)) = \mathsf{A}}{\Gamma \vdash x : \Gamma(x) \;;\; \Gamma[x \mapsto \bullet]} \qquad \textsc{LitB}\ \frac{}{\Gamma \vdash b_\ell : \mathtt{bit}^{\bot}_{\ell} \;;\; \Gamma}$$

$$\textsc{LitN}\ \frac{}{\Gamma \vdash n_\ell : \mathtt{nat}^{\bot}_{\ell} \;;\; \Gamma} \qquad \textsc{AsN}\ \frac{\Gamma \vdash p : \mathtt{bit}^{\rho}_{\ell} \;;\; \Gamma'}{\Gamma \vdash \mathtt{asnat}(p) : \mathtt{nat}^{\rho}_{\ell} \;;\; \Gamma'} \qquad \textsc{AsB}\ \frac{\Gamma \vdash p : \mathtt{nat}^{\rho}_{\ell} \;;\; \Gamma'}{\Gamma \vdash \mathtt{asbit}(p) : \mathtt{bit}^{\rho}_{\ell} \;;\; \Gamma'}$$

$$\textsc{Flip}\ \frac{\rho \neq \bot}{\Gamma \vdash \mathtt{flip}^{\rho}() : \mathtt{flip}^{\rho} \;;\; \Gamma} \qquad \textsc{Comp}\ \frac{\Gamma \vdash p : \mathtt{flip}^{\rho} \;;\; \Gamma'}{\Gamma \vdash p^{\circ} : \mathtt{flip}^{\rho} \;;\; \Gamma'} \qquad \textsc{Use}\ \frac{\Gamma(x) = \mathtt{flip}^{\rho}}{\Gamma \vdash \mathtt{use}(x) : \mathtt{bit}^{\rho}_{\mathsf{S}} \;;\; \Gamma}$$

$$\textsc{Rev}\ \frac{\Gamma(x) = \mathtt{flip}^{\rho}}{\Gamma \vdash \mathtt{reveal}(x) : \mathtt{bit}^{\bot}_{\mathsf{P}} \;;\; \Gamma[x \mapsto \bullet]} \qquad \textsc{Aop}\ \frac{\Gamma \vdash p_1 : \mathtt{nat}^{\rho_1}_{\ell_1} \;;\; \Gamma' \qquad \Gamma' \vdash p_2 : \mathtt{nat}^{\rho_2}_{\ell_2} \;;\; \Gamma''}{\Gamma \vdash p_1 \odot p_2 : \mathtt{nat}^{\rho_1 \sqcup \rho_2}_{\ell_1 \sqcup \ell_2} \;;\; \Gamma''}$$

$$\textsc{MuxBit}\ \frac{\Gamma \vdash p_1 : \mathtt{bit}^{\rho_1}_{\ell_1} \;;\; \Gamma' \qquad \Gamma' \vdash p_2 : \mathtt{bit}^{\rho_2}_{\ell_2} \;;\; \Gamma'' \qquad \Gamma'' \vdash p_3 : \mathtt{bit}^{\rho_3}_{\ell_3} \;;\; \Gamma'''}{\Gamma \vdash \overset{\mathsf{U}}{\mathtt{mux}}(p_1, p_2, p_3) : \mathtt{bit}^{\rho_1 \sqcup \rho_2 \sqcup \rho_3}_{\ell_1 \sqcup \ell_2 \sqcup \ell_3} \times \mathtt{bit}^{\rho_1 \sqcup \rho_2 \sqcup \rho_3}_{\ell_1 \sqcup \ell_2 \sqcup \ell_3} \;;\; \Gamma'''}$$

$$\textsc{MuxFlip}\ \frac{\Gamma \vdash p_1 : \mathtt{bit}^{\rho_1}_{\ell} \;;\; \Gamma' \qquad \Gamma' \vdash p_2 : \mathtt{flip}^{\rho_2} \;;\; \Gamma'' \qquad \Gamma'' \vdash p_3 : \mathtt{flip}^{\rho_3} \;;\; \Gamma''' \qquad \rho_1 \perp\!\!\!\perp \rho_2 \qquad \rho_1 \perp\!\!\!\perp \rho_3}{\Gamma \vdash \overset{\mathsf{A}}{\mathtt{mux}}(p_1, p_2, p_3) : \mathtt{flip}^{\rho_2 \sqcup \rho_3} \times \mathtt{flip}^{\rho_2 \sqcup \rho_3} \;;\; \Gamma'''}$$

$$\textsc{Tup}\ \frac{\Gamma \vdash p_1 : \tau_1 \;;\; \Gamma' \qquad \Gamma' \vdash p_2 : \tau_2 \;;\; \Gamma''}{\Gamma \vdash \langle p_1, p_2 \rangle : \tau_1 \times \tau_2 \;;\; \Gamma''} \qquad \textsc{Arr}\ \frac{\Gamma_{n-1} \vdash p_n : \tau \;;\; \Gamma_n}{\Gamma_0 \vdash \mathtt{array}^{N}(p_1, .., p_N) : \mathtt{array}(\tau) \;;\; \Gamma_N}$$

$$\textsc{Read}\ \frac{\Gamma \vdash p_1 : \mathtt{array}(\tau) \;;\; \Gamma' \qquad \Gamma' \vdash p_2 : \mathtt{nat}^{\bot}_{\mathsf{P}} \;;\; \Gamma'' \qquad \mathcal{K}(\tau) = \mathsf{U}}{\Gamma \vdash p_1[p_2] : \tau \;;\; \Gamma''} \qquad \textsc{Write}\ \frac{\Gamma \vdash p_1 : \mathtt{array}(\tau) \;;\; \Gamma' \qquad \Gamma' \vdash p_2 : \mathtt{nat}^{\bot}_{\mathsf{P}} \;;\; \Gamma'' \qquad \Gamma'' \vdash p_3 : \tau \;;\; \Gamma'''}{\Gamma \vdash p_1[p_2] \leftarrow p_3 : \tau \;;\; \Gamma'''}$$

$$\textsc{Let}\ \frac{\Gamma \vdash a : \tau_1 \;;\; \Gamma' \qquad \Gamma' \uplus [x \mapsto \tau_1] \vdash e : \tau_2 \;;\; \Gamma'' \uplus [x \mapsto \_]}{\Gamma \vdash \mathtt{let}\ x \coloneqq a\ \mathtt{in}\ e : \tau_2 \;;\; \Gamma''}$$

$$\textsc{LetT}\ \frac{\Gamma \vdash a : \tau_1 \times \tau_2 \;;\; \Gamma' \qquad \Gamma' \uplus [x_1 \mapsto \tau_1, x_2 \mapsto \tau_2] \vdash e : \tau_3 \;;\; \Gamma'' \uplus [x_1 \mapsto \_, x_2 \mapsto \_]}{\Gamma \vdash \mathtt{let}\ x_1, x_2 \coloneqq a\ \mathtt{in}\ e : \tau_3 \;;\; \Gamma''}$$

$$\textsc{If}\ \frac{\Gamma \vdash p : \mathtt{bit}^{\bot}_{\mathsf{P}} \;;\; \Gamma' \qquad \Gamma' \vdash e_1 : \tau_1 \;;\; \Gamma_1'' \qquad \Gamma' \vdash e_2 : \tau_2 \;;\; \Gamma_2''}{\Gamma \vdash \mathtt{if}\ p\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau_1 \sqcup \tau_2 \;;\; \Gamma_1'' \sqcup \Gamma_2''}$$

Fig. 7. Typing Judgment for $L_{\mathbf{obliv}}$

probabilistic dependencies between values. Regions must be a distributive lattice with a least element $\bot$. A distributive lattice must support both join $\sqcup$ and meet $\sqcap$ operations, and meets must distribute through joins, that is: $\rho_1 \sqcap (\rho_2 \sqcup \rho_3) = (\rho_1 \sqcap \rho_2) \sqcup (\rho_1 \sqcap \rho_3)$. We notate the meet of two regions being bottom $\rho_1 \sqcap \rho_2 = \bot$ as $\rho_1 \perp\!\!\!\perp \rho_2$, which we call *region independence*. Distributivity is necessary for a conjunction of region independence to coincide with region independence for the join, that is: $\rho_1 \perp\!\!\!\perp \rho_2 \wedge \rho_1 \perp\!\!\!\perp \rho_3 \iff \rho_1 \perp\!\!\!\perp (\rho_2 \sqcup \rho_3)$.

Rule Flip types a coin flip. The region annotation ρ is a static name for coin flips generated at this (so-annotated) program point. The region must not be $\bot$ because $\bot$ represents independence from all random values, and a coin flip is dependent on itself. When created, a coin flip is independent from any other value with an independent region, so ρ' where $\rho \perp\!\!\!\perp \rho'$, and that the distribution is *stable*, meaning each possible bit value `0` and `I` has equal probability. Given this interpretation of the flip type, we are able to typecheck the bit-complement of a value at flip type as also having flip type, given as rule Comp.

Rules for literals LitB and LitN track the security label ℓ of the literal, and are given probability region $\bot$ to signify they are probabilistically independent of all other random values. Rules Use and Reveal, mentioned above, crucially preserve the probability region ρ from the input $\texttt{flip}^{\rho}$ on the output type. As such, the type system can track their potential dependence on other values. Rules AsN and AsB convert between `bit` and `nat` types, also preserving the region. Per rule Aop, the result type of an arithmetic operation is given the join of operand labels and the join of operand regions. In particular, joining the regions indicates that whatever correlates with p_1 or p_2 may also correlate with the result of the operation, which is a safe over-approximation of correlation for the result of any binary arithmetic operation which operates directly on values (*i.e.*, not distributions of values).

A critical feature of the type system is the handling of `mux` expressions, which are treated by rules MuxBit and MuxFlip for bit and flip types, respectively. The mux rule for bits treats mux as a ternary bit operation, and takes the join of security labels and probability regions of each argument. The mux rule for flip types is designed to avoid passing on any probabilistic dependence from the guard p_1 to the results—notice that the labels on the output pair of flip types are $\rho_2 \sqcup \rho_3$ and not $\rho_1 \sqcup \rho_2 \sqcup \rho_3$. This is acceptable when the probability region ρ_1 of the guard p_1 is independent of the regions of the arguments, i.e., $\rho_1 \perp\!\!\!\perp \rho_2$ and $\rho_1 \perp\!\!\!\perp \rho_3$. As such, any revelation of values in ρ_1 will not affect the stability or independence of ρ_2 or ρ_3.

For example, the following program is well-typed:

```
1  let x = rnd^ρ0 in
2  let y = rnd^ρ in
3  let z = rnd^ρ in
4  let s_x = use(x) in
5  let (r1,r2) = mux (s_x = 0) y z in
6  output (reveal x);
```

According to the MuxFlip rule, both `r1` and `r2` have type $\texttt{flip}^{\rho}$—there is no influence from `x`'s region ρ_0. This is as it should be: because `y` and `z` are probabilistically independent from `x`, the revelation of `x` on line 6 does not affect their stability. On other hand, rule MuxFlip prevents the second problematic example from Section 3.4 (page 9). In that example the **mux** on line 3 is disallowed because the region of the guard `s_x = 0` is not independent of the region of its argument `x`—they are the same.

The remaining type rules handle tuples, arrays, let binding, and normal conditionals; we remark on a few important features of them. First, notice that Read and Write both require that the array index p_2 has label `P`; this is important because we assume the adversary is able to observe array accesses. (It also avoids leaks due to programs accessing the array out of bounds.) Rule If similarly requires the guard p to have label `P` since the execution trace reveals which branch is taken. Second,

both Let and LetT remove their bound variables from the output environment; we write $\Gamma'' \uplus [x \mapsto \_]$ to split a context into the part that binds x and Γ'' binds the rest; it is the latter part that is returned, dropping the x binding.

5 PROBABILISTIC MEMORY TRACE OBLIVIOUSNESS

This section describes our proof of memory trace obliviousness for $L_{\mathbf{obliv}}$. We begin by presenting a model for distributions, probabilities, conditional probabilities, and probabilistic dependence. Next we give a denotational semantics based on this model, and which we prove coincides with the sampling semantics in Section 4.2. Equipped with a model for distributions and a denotational semantics we then state precisely the definition of memory trace obliviousness. To describe our proof approach, we proceed in more informal terms and describe the key semantic invariants which hold of the denotational semantics. These key invariants are sufficient to prove a strengthened memory trace obliviousness property, and this strengthened property implies the simpler property initially stated. Further details of the proof, including precise mathematical definitions and properties, are given in the Appendix.

5.1 Distributions

We use a discrete model for distributions which are defined over a fixed universe with finite entropy. In our model, a distribution of elements $x \in X$ in universe R is notated $\tilde{x}^R$. Here, R is the number of available coin flips. So distributions are represented as a mapping from a bitvector $\vec{b}$ of length R to an element of X, that is: $\tilde{x}^R \in \mathbb{B}^R \to X$. This domain supports modeling the result of a terminating computation which flips an arbitrary, finite number of coins. Our model does not support nonterminating computations, especially those which may flip an infinite number of coins. As an example, suppose we want $\tilde{y}^R$ to represent the distribution of the count of coin flips that have come up 0. Thus $\tilde{y}^R$ is a function that takes a length-R vector of coin flips and returns a natural number that sums the occurrences of 0 in the vector.

We define a probability measure for distributions $\mathfrak{p}\left[\tilde{x}^R = x\right]$ by counting the occurrences of x in $\tilde{x}^R$ and dividing by 2^R. So for our example, the probability $\mathfrak{p}\left[\tilde{y}^R = 2\right]$ when $R = 2$ would be $\frac{1}{4}$. The model $\mathbb{B}^R \to X$ for distributions is particularly useful for expressing a measure for conditional probability. Conditional probabilities for distributions $\mathfrak{p}\left[\tilde{x}^R = x \;\middle|\; \tilde{y}^R = y\right]$ are defined by counting simultaneous occurrences of x in $\tilde{x}^R$ and y in $\tilde{y}^R$ and dividing by occurrences of y in $\tilde{y}^R$. Conditional probabilities are therefore a *derived* notion, as opposed to an axiomatic notion, as is typically the case when modeling distributions as $\tilde{x} \in X \to [0..1]$. When the universe R is not informative, we will omit it and just write $\tilde{x}$. We notate joint probabilities as $\mathfrak{p}\,[\tilde{x} = x \wedge \tilde{y} = y]$. We notate point distributions $\lfloor x \rfloor$ where $\mathfrak{p}\,[\lfloor x \rfloor = x] = 1$.

Key to our approach is careful bookkeeping of a set of coin flips $\theta \in \wp(\mathbb{N}_{\leq R})$ upon which all information observable to the adversary depends. We notate probabilistic dependency $\tilde{x} \ltimes \theta$ and prove that if two random variables $\tilde{x}$ and $\tilde{y}$ depend on disjoint sets of coin flips $\theta_1 \cap \theta_2 = \varnothing$ then they are probabilistically independent, written $\tilde{x} \perp\!\!\!\perp \tilde{y}$, which means their joint probability factors: $\mathfrak{p}\,[\tilde{x} = x \wedge \tilde{y} = y] = \mathfrak{p}\,[\tilde{x} = x]\,\mathfrak{p}\,[\tilde{y} = y]$. We will often notate a probability conditioned on some θ as $\mathfrak{p}\,[\tilde{x} = x \mid \theta] = q$, which is the probability that $\tilde{x} = x$ no matter the values of each of the coin flips in θ. We notate probabilistic dependency conditioned on a set of coin flips θ modulo θ' as $\tilde{x} \ltimes_{/\!/\theta'} \theta$, and probabilistic independence modulo θ as $\tilde{x} \perp\!\!\!\perp_{/\!/\theta} \tilde{y}$.

We call two distributions $\tilde{x}_1$ and $\tilde{x}_2$ equivalent, notated $\tilde{x}_1 \approx \tilde{x}_2$, when their probability measures coincide for all elements, that is $\forall x.\ \ \mathfrak{p}\,[\tilde{x}_1 = x] = [\tilde{x}_2 = x]$. We call $\tilde{x}_1$ and $\tilde{x}_2$ equivalent modulo

θ, notated $\tilde{x}_1 \approx_{/\!/\theta} \tilde{x}_2$, when their probability measures coincide for all elements conditioned on θ, that is $\forall x.\ \mathfrak{p}\,[\tilde{x}_1 = x \mid \theta] = \mathfrak{p}\,[\tilde{x}_2 = x \mid \theta]$.

Another key invariant in our proof will regard a distribution's *stability*, notated $\mathit{stable}\,[\tilde{x}]$. Stability is the property that each outcome of a distribution is equally likely, that is $\forall x_1\ x_2.\ \mathfrak{p}\,[\tilde{x} = x_1] = \mathfrak{p}\,[\tilde{x} = x_2]$. We call a distribution $\tilde{x}$ stable modulo θ, notated $\mathit{stable}\,[\tilde{x} \mid \theta]$, when it is stable conditioned on θ, that is $\forall x_1\ x_2.\ \mathfrak{p}\,[\tilde{x} = x_1 \mid \theta] = \mathfrak{p}\,[\tilde{x} = x_2 \mid \theta]$.

5.2 Denotational Semantics

To facilitate the proof of memory trace obliviousness for $L_{\mathbf{obliv}}$, we define a denotational semantics over distributions, and which we prove corresponds with the sampling semantics defined previously in Section 4. We define the denotational semantics in two parts, an evaluation semantics for expressions, notated $\mathcal{E}[\![\cdot]\!]$, and a trace semantics for expressions, notated $\mathcal{T}[\![\cdot]\!]$. The evaluation semantics $\mathcal{E}[\![\cdot]\!]$ denotes an expression e to a partial function from distributions of environments $\tilde{\gamma}$ and stores $\tilde{\sigma}$ to distributions of values $\tilde{v}$, environments $\tilde{\gamma}'$ and stores $\tilde{\sigma}'$. Recall that a configuration ς is an expression, a (non-distributional) environment, and a store, while an execution trace t is a sequence of configurations. The trace semantics $\mathcal{T}[\![\cdot]\!]$ then denotes an expression to a partial function from distributions of environments $\tilde{\gamma}$ and stores $\tilde{\sigma}$ to distributions of traces $\tilde{t}$.

$$\mathcal{E}[\![\cdot]\!] : \mathit{exp} \to \widetilde{\mathit{env}} \times \widetilde{\mathit{store}} \rightharpoonup \widetilde{\mathcal{V}} \times \widetilde{\mathit{env}} \times \widetilde{\mathit{store}}$$
$$\mathcal{T}[\![\cdot]\!] : \mathit{exp} \to \widetilde{\mathit{env}} \times \widetilde{\mathit{store}} \rightharpoonup \widetilde{\mathit{config}^*}$$

The semantics is partial due to the possibility of runtime type errors. However, type safety (proved later) guarantees that the denotational semantics is total for well-typed expressions.

Theorem 5.1 (Sampling and Denotation Correspondence). *The sampling semantics $\varsigma \rightsquigarrow^q \varsigma$ and the denotational semantics $\mathcal{T}[\![\cdot]\!]$ correspond, that is for $\varsigma_1 := \langle e_1, \gamma_1, \sigma_1 \rangle$ and $\varsigma_N := \mathsf{HALT}(v, \gamma_N, \sigma_N)$:*

$$\begin{array}{rl} \textit{If:} & \varsigma_1 \rightsquigarrow^{q_1} \cdots \rightsquigarrow^{q_N} \varsigma_N \quad \textit{and} \quad \tilde{t} := \mathcal{T}[\![e_1]\!](\lfloor\gamma_1\rfloor, \lfloor\sigma_1\rfloor) \\ \textit{Then:} & \mathfrak{p}\left[\tilde{t} = \varsigma_1 \ldots \varsigma_N\right] = \prod_{n \in \{1..n\}} q_n \end{array}$$

5.3 Memory Trace Obliviousness

The design of $L_{\mathbf{obliv}}$ is such that well typed programs have the property of Memory Trace Obliviousness (MTO). We state and prove MTO with respect to an adversary that is able to observe the memory and instruction access pattern of the execution. The MTO theorem establishes that the adversary learns nothing from these observations.

Theorem 5.2 (Memory Trace Obliviousness (MTO)). *Given a well typed $L_{\mathbf{obliv}}$ expression e and two value environments γ_1 and γ_2 which are both well-formed and agree on public values, then the publicly observable traces of executing e using γ_1 or γ_2 yields equivalent distributions, that is:*

$$\begin{array}{rl} \textit{If:} & \Gamma \vdash e : \tau\ ;\ \Gamma' \quad \textit{and} \quad \Gamma \vdash \gamma_1 \quad \textit{and} \quad \Gamma \vdash \gamma_2 \quad \textit{and} \quad \mathit{obs}(\gamma_1) = \mathit{obs}(\gamma_2) \\ \textit{Then:} & \widetilde{\mathit{obs}}(\mathcal{T}[\![e]\!](\lfloor\gamma_1\rfloor, \varnothing)) \approx \widetilde{\mathit{obs}}(\mathcal{T}[\![e]\!](\lfloor\gamma_2\rfloor, \varnothing)) \end{array}$$

We define the public observations of a value, environment, store or configuration using obs (see Section 4.2), and notate its lifting to distributions as $\widetilde{\mathit{obs}}$.

5.4 Proof outline

This section outlines the key ideas and invariants required for the proof of MTO. We provide further mathematical detail of definitions and key properties in the Appendix.

The first property we prove is type safety, which shows that the partial denotation functions are actually total under the assumption that the term is well typed.

Theorem 5.3 (Type Safety). *Given a well-typed $L_{\textbf{obliv}}$ expression e, and distributions of value environments $\tilde{\gamma}$ and value stores $\tilde{\sigma}$ which are well-formed, then the evaluation semantics $\mathcal{E}[\![e]\!]$ is always defined, and the result is well-formed, that is:*

$$\begin{array}{rl} \textit{If:} & \Gamma \vdash e : \tau \;;\; \Gamma' \quad \textit{and} \quad \Gamma \;;\; \Sigma \vdash \tilde{\gamma} \quad \textit{and} \quad \Sigma \vdash \tilde{\sigma} \\ \textit{Then there exists:} & \Sigma', \tilde{v}, \tilde{\gamma}' \textit{ and } \tilde{\sigma}' \\ \textit{Such that:} & \mathcal{E}[\![e]\!](\tilde{\gamma}, \tilde{\sigma}) = \langle \tilde{v}, \tilde{\gamma}', \tilde{\sigma}' \rangle \quad \textit{and} \quad \Sigma' \vdash \tilde{v} \quad \textit{and} \quad \Gamma' \;;\; \Sigma' \vdash \tilde{\gamma}' \quad \textit{and} \quad \Sigma' \vdash \tilde{\sigma}' \end{array}$$

We then prove type safety for the trace semantics as a consequence of type safety for the evaluation semantics.

The next step in the proof is to:

(1) Give a semantics to regions which appear in types; and
(2) Relate this semantics for regions to the evaluation semantics

We achieve (1) by defining a *region semantics*, notated $\mathcal{R}[\![\cdot]\!]$, which when applied to a configuration ς, produces a map from regions ρ to the concrete set of coin flips θ that occurred at a $\texttt{flip}^{\rho}()$ (i.e., with that region annotation) during the evaluation of ς. We call this map a *region model*, notated ξ. A key invariant of the region model is that disjoint regions map to disjoint sets, that is: $\forall \rho_1\ \rho_2.\ \ \rho_1 \sqcap \rho_2 = \bot \Rightarrow \xi(\rho_1) \cap \xi(\rho_2) = \varnothing$.

We achieve (2) in two parts. First, we first define a *dependency semantics*, notated $\mathcal{D}[\![\cdot]\!]$, which maps configurations to (in the case of base types) a set of coin flips θ. This set *over*-approximates the ground-truth dependencies of the resulting value $\tilde{v}$ after conditioning on public revelations, which is given as a set of coin flip dependencies Φ, that is $\tilde{v} \ltimes_{/\!/\Phi} \theta$. The desired relationship between regions and values is that whenever an expression annotated with region ρ evaluates to a value $\tilde{v}$, then the dependency set θ predicted by the dependency semantics $\mathcal{D}[\![\cdot]\!]$ is included in the set of coin flips associated with the region model, that is $\tilde{v} \ltimes_{/\!/\Phi} \theta \subseteq \xi(\rho)$. We make each of these steps precise as theorems used in the (final) proof.

Theorem 5.4 (Dependency Soundness). *Given a well-typed $L_{\textbf{obliv}}$ expression e and distributions of value environments $\tilde{\gamma}$ and value stores $\tilde{\sigma}$ which are well-formed w.r.t. models for dependencies in the environment and store respectively, conditioned on the set of publicly revealed bits Φ, then the result of the dependency semantics $\mathcal{D}[\![\cdot]\!]$ is well-formed w.r.t. the results of the evaluation semantics $\mathcal{E}[\![\cdot]\!]$ conditioned on some equal or larger set of publicly revealed bits Φ'.*

Theorem 5.5 (Region Soundness). *Given a well-typed $L_{\textbf{obliv}}$ expression e and distributions of value environments $\tilde{\gamma}$ and value stores $\tilde{\sigma}$ which are well-formed w.r.t. a region model ξ conditioned on the set of publicly revealed bits Φ, then the result of the region semantics $\mathcal{R}[\![\cdot]\!]$ is well-formed w.r.t. the results of the dependency semantics $\mathcal{D}[\![\cdot]\!]$ conditioned on some equal or larger set of publicly revealed bits Φ'.*

The primary purpose of the region semantics, the dependency semantics, and their soundness properties is to establish that when terms are typed at disjoint regions $\rho_1 \perp\!\!\!\perp \rho_2$, then their computed values will be probabilistically independent $\tilde{v}_1 \perp\!\!\!\perp \tilde{v}_2$. We now turn to the two key invariants that justify memory trace obliviousness in the presence of $\texttt{reveal}(p)$ expressions: *partitioning* and *stability*.

We say that evaluation contexts $\langle \tilde{\gamma}, \tilde{\sigma} \rangle$ are *well-partitioned* if and only if any two bit distribution values $\tilde{b}_1$ and $\tilde{b}_2$ at type `flip` occurring at different positions in the evaluation context are guaranteed to be independent of each other, conditioned on publicly revealed information Φ, that is $\tilde{b}_1 \perp\!\!\!\perp_{/\!/\Phi} \tilde{b}_2$. We prove a theorem that the evaluation semantics preserves well partitioning, conditioned on public information.

Theorem 5.6 (Well-partitioning). *Given a well-typed $L_{\textbf{obliv}}$ expression e and distributions of value environments $\tilde{\gamma}$ and value stores $\tilde{\sigma}$ where every contained* `flip` *value is well-partitioned w.r.t. other* `flip` *values conditioned on the set of publicly revealed bits Φ, then the result of the evaluation semantics $\mathcal{E}[\![\cdot]\!]$ is well-partitioned for some equal or larger set of publicly revealed bits Φ'.*

The key part of the proof of well-partitioning is for `reveal`(p) expressions. Before the revelation, it is assumed all flip values are well partitioned. Immediately after the reveal expressions, flip values will remain well-partitioned, but conditioned on a larger set of publicly revealed bits Φ', to which the publicly revealed flip is added. In subsequent execution, the publicly revealed bit may end up correlating with flip values (*e.g.* as a result of a public `if` expression), however because partitioning is quotiented by publicly revealed bits, this correlation ends up being benign.

The mechanism in the type system that enforces this semantic property of well-partitioning is the affine treatment of values at type `flip`; because `flip` values cannot be duplicated, there will always only be one copy of each coin flip in the context, and therefore each flip value will be independent of other flip values.

We define *stability* also as a property of evaluation contexts $\langle \tilde{\gamma}, \tilde{\sigma} \rangle$, such that any two bit distribution values $\tilde{b}$ at type `flip` in the evaluation context (either $\tilde{\gamma}$ or $\tilde{\sigma}$) are guaranteed to be stable distributions conditioned on publicly revealed information, that is: $\mathit{stable}\left[\tilde{b} \;\middle|\; \Phi\right]$. We prove a theorem that the evaluation semantics $\mathcal{E}[\![\cdot]\!]$ preserves stability, again conditioned on public information.

Theorem 5.7 (Stability). *Given a well-typed $L_{\textbf{obliv}}$ expression e and distributions of value environments $\tilde{\gamma}$ and value stores $\tilde{\sigma}$ where every contained* `flip` *value is stable conditioned on the set of publicly revealed bits Φ, then the result of the evaluation semantics $\mathcal{E}[\![\cdot]\!]$ is stable conditioned for some equal or larger set of publicly revealed bits Φ'.*

The proof of stability uses the proof of partitioning as a lemma. The key part of the proof is for `reveal`(p) expressions. Before the revelation, it is assumed all flip values are stable. Immediately after the revelation, it must be shown that all other coin flips remain stable. This is a consequence of well-partitioning, which guarantees that the revealed coin flip will not correlate with any other coin flip, and therefore adding the revealed bit to the set of public information does not further perturb the remaining stable coin flips in the context.

The mechanism in the type system which enforces the semantic property of stability is the typing rules for bit complement and mux. Bit complement is safe w.r.t. stability, which we prove, and the mux rule is stable under the conditions of region independence, which in turn imply that the underlying distributions of values are probabilistically independent. Other than public revelation, these are the only two language forms which manipulate coin flips, and their typing rules guarantee that resulting coin flips will remain stable.

In order to prove the desired memory trace obliviousness property, we state a stronger property which implies the desired property, but which supports a direct proof by induction on the term. This stronger trace obliviousness property includes all of the assumptions for each of the above lemmas, so each lemma can be applied in each case of the proof by induction, for which we show the high-level details.

Theorem 5.8 (Strong Memory Trace Obliviousness (SMTO)). *Given a well-typed $L_{\textbf{obliv}}$ expression e and satisfying properties from the above lemmas applied to each of two sets of distribution value environments and stores $\langle \tilde{\gamma}_1, \tilde{\sigma}_1 \rangle$ and $\langle \tilde{\gamma}_2, \tilde{\sigma}_2 \rangle$, and where each of $\tilde{\gamma}_1$ and $\tilde{\gamma}_2$ and $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$ are distributed equally for publicly observable values conditioned on publicly revealed bits Φ (i.e., $\widetilde{obs}(\tilde{\gamma}_1) \approx_{/\!/\Phi} \widetilde{obs}(\tilde{\gamma}_2)$ and $\widetilde{obs}(\tilde{\sigma}_1) \approx_{/\!/\Phi} \widetilde{obs}(\tilde{\sigma}_2)$), then the observable traces for e applied to either set of contexts $\langle \tilde{\gamma}_1, \tilde{\sigma}_1 \rangle$ or $\langle \tilde{\gamma}_2, \tilde{\sigma}_2 \rangle$ will yield identical distributions conditioned on publicly revealed bits Φ (i.e., $\widetilde{obs}(\mathcal{T}[\![e]\!](\tilde{\gamma}_1, \tilde{\sigma}_1)) \approx_{/\!/\Phi} \widetilde{obs}(\mathcal{T}[\![e]\!](\tilde{\gamma}_2, \tilde{\sigma}_2))$).*

Proof. We prove this theorem by induction on the syntax for terms, and by applying each other theorem developed thus far. For example, for `let` expressions, the proof goal is of the form:

$$\widetilde{obs}(\mathcal{T}[\![\texttt{let } x := a \texttt{ in } e]\!](\tilde{\gamma}_1, \tilde{\sigma}_1)) \approx_{/\!/\Phi} \widetilde{obs}(\mathcal{T}[\![\texttt{let } x := a \texttt{ in } e]\!](\tilde{\gamma}_2, \tilde{\sigma}_2))$$

From the definition of the trace semantics $\mathcal{T}[\![\cdot]\!]$, this goal becomes:

$$\lfloor e \rfloor \otimes \widetilde{obs}(\tilde{\gamma}_1'') \otimes \widetilde{obs}(\tilde{\sigma}_1') \mathbin{\widetilde{+\!\!+}} \widetilde{obs}(\mathcal{T}[\![e]\!](\tilde{\gamma}_1'', \tilde{\sigma}_1')) \approx_{/\!/\Phi} \lfloor e \rfloor \otimes \widetilde{obs}(\tilde{\gamma}_2'') \otimes \widetilde{obs}(\tilde{\sigma}_2') \mathbin{\widetilde{+\!\!+}} \widetilde{obs}(\mathcal{T}[\![e]\!](\tilde{\gamma}_2'', \tilde{\sigma}_2'))$$

where $\langle \tilde{v}, \tilde{\gamma}_i', \tilde{\sigma}_i' \rangle := \mathcal{E}[\![a]\!](\tilde{\gamma}_i, \tilde{\sigma}_i)$, $\tilde{\gamma}_i'' := \tilde{\gamma}_i'[x \mapsto \tilde{v}]$, and $\otimes$ and $\mathbin{\widetilde{+\!\!+}}$ are product and concatenation operations respectively lifted to distributions. This goal is broken down using the product chain rule for distributions, which states, using our notation and in the case of three elements:

$$\mathfrak{p}[\tilde{x} \otimes \tilde{y} \otimes \tilde{z} = \langle x, y, z \rangle] = \mathfrak{p}[\tilde{x} = x]\mathfrak{p}[\tilde{y} = y \mid \tilde{x} = x]\mathfrak{p}[\tilde{z} = z \mid \tilde{x} = x \wedge \tilde{y} = y]$$

In the case that two elements of a tuple are independent, then it factors without conditioning, as per the definition of independence.

Using the product chain rule, in order to show the two distributions in the goal are equivalent, it suffices to show each component of the three-tuple is independent of other elements of the three-tuple (*e.g.* $\widetilde{obs}(\tilde{\gamma}_1'') \perp\!\!\!\perp \widetilde{obs}(\tilde{\sigma}_1')$), that each element of the three-tuple is equivalent to its pairwise element (*e.g.* $\widetilde{obs}(\tilde{\gamma}_1'') \approx_{/\!/\Phi} \widetilde{obs}(\tilde{\gamma}_2'')$), and that the recursive occurrence of the trace is equivalent pairwise conditioned on the three-tuple. That is, after establishing independence of elements in the three-tuple, it suffices to show:

(1) $\mathfrak{p}\left[\widetilde{obs}(\tilde{\gamma}_1'')\right] \approx_{/\!/\Phi} \mathfrak{p}\left[\widetilde{obs}(\tilde{\gamma}_2'')\right]$
(2) $\mathfrak{p}\left[\widetilde{obs}(\tilde{\sigma}_1')\right] \approx_{/\!/\Phi} \mathfrak{p}\left[\widetilde{obs}(\tilde{\gamma}_2'')\right]$
(3) $\mathfrak{p}\left[\widetilde{obs}(\mathcal{T}[\![e]\!](\tilde{\gamma}_1'', \tilde{\sigma}_1')) \;\middle|\; \widetilde{obs}(\tilde{\gamma}_1'') \wedge \widetilde{obs}(\tilde{\sigma}_1')\right] \approx_{/\!/\Phi} \mathfrak{p}\left[\widetilde{obs}(\mathcal{T}[\![e]\!](\tilde{\gamma}_2'', \tilde{\sigma}_2')) \;\middle|\; \widetilde{obs}(\tilde{\gamma}_2'') \wedge \widetilde{obs}(\tilde{\sigma}_2')\right]$

Goals (1) and (2) are consequences of partitioning and stability theorems. Goal (3) is further refined based on the observation that the new contexts $\langle \tilde{\gamma}_1'', \tilde{\sigma}_1' \rangle$ and $\langle \tilde{\gamma}_2'', \tilde{\sigma}_2' \rangle$ depend uniquely on an equal or larger set of publicly revealed bits Φ' under which the contexts remain well-partitioned and stable. It is therefore sufficient to condition on this (potentially) larger set of publicly revealed bits Φ', and demonstrate equivalence of distributions module Φ', that is:

$$\mathfrak{p}\left[\widetilde{obs}(\mathcal{T}[\![e]\!](\tilde{\gamma}_1'', \tilde{\sigma}_1')) \;\middle|\; \Phi'\right] \approx_{/\!/\Phi'} \mathfrak{p}\left[\widetilde{obs}(\mathcal{T}[\![e]\!](\tilde{\gamma}_2'', \tilde{\sigma}_2')) \;\middle|\; \Phi'\right]$$

At this point, the inductive hypothesis applies and the proof for `let` is complete. The proofs for `let` on tuples and public `if` statements follow the same structure, and are analogous. □

After proving SMTO, we recover a proof of MTO as an instantiation of SMTO with the empty store $\widehat{\sigma} = \lfloor \varnothing \rfloor$ and the empty set of publicly revealed bits $\Phi = \{\}$.

6 CASE STUDY

We have implemented a type checker for an extension to $L_{\mathbf{obliv}}$ that adds functions, (bounded) recursion, and various arithmetic operations. We have used it to check two classical probabilistically oblivious algorithms: non-recursive ORAM (NORAM) and tree-based ORAMs. These were first mentioned in Section 3.2, where we used NORAM as a building block to implement an oblivious stack. In this section, we discuss the implementation of both NORAM and tree ORAM, where the latter makes use of the former.

In what follows, we write `nat S` to be a secret number in region $\bot$; `nat P` for a public number in $\bot$; `nat R` to be a secret in some region `R`. We also write type `rnd R` to be a random natural number in region `R`; in our implementation we encode these using tuples of `flip R`s.

6.1 Non-recursive ORAM

A non-recursive ORAM (NORAM) is the foundation all tree-based ORAMs. Such NORAMs provide the state-of-the-art in performance, and enable efficient oblivious data structures. A NORAM organizes data blocks as a complete tree of depth $\log N$ where N is the capacity of the ORAM. Each node on the tree is a *bucket*, which is essentially a *trivial ORAM* of size b, which we call *the bucket size*. A trivial ORAM is basically an array in which reads and writes require iterating through each element in the array; we showed the trivial ORAM `add` operation in Section 2.2. Each NORAM data block bucket associated with a randomly generated *position tag*, which is a $\log N$-bit random number, corresponding to one leaf of the tree. A key invariant of a tree-based NORAM is that each data block always resides in a bucket somewhere on the path from the root to the leaf corresponding to its position tag.

In particular, we have the following signature:

```
type noram R R' = (bucket R R') array
type bucket = (block R R') array
type block R R' = bit R * nat R * nat R * (rnd R' * rnd R') where R ⫫ R'
```

A NORAM is an array of $2N-1$ buckets which represent a complete tree in the style of a heap data structure: for the node at index $i \in \{0, \ldots, 2N-2\}$, its parents, left child, and right child correspond to the nodes at index $(i-1)/2$, $2i+1$, and $2i+2$, respectively. Each bucket is an array of blocks, each of which is a 4-tuple, where the last element is the actual data. Ideally, an NORAM should be able to store data of an arbitrary type α, but since our language does not yet support polymorphism we fix the data to type `rnd R' * rnd R'` where `R` and `R'` are independent. We choose this type to illustrate the affine values can be stored in the NORAM, and to set up our implementation of Tree ORAM next. (In our actual interpreter, we duplicate the code for each type we wish to store.) The other three components of the block are all secret and in region `R`; they are (1) a bit indicating whether the block is dummy or not; (2) the index of the block; and (3) the position tag for the block.

NORAM supports two primitives, `rr` and `add`, which we introduced (in greater generality) in Section 3.2. Their signatures for our implementation are:

```
rr:  noram R R' → (idx:nat R)  → (tag:nat P) → (data:rnd R' * rnd R')
add: noram R R' → (idx:nat R) → (tag:nat R) → (data:rnd R' * rnd R') → unit
```

Due to the NORAM invariant, when the position tag corresponding to a block is known, `rr` can be implemented as a linear scan through the path corresponding to the tag to find the block with the given index. In doing so, $\log N$ buckets are accessed, each with b blocks. Therefore, the `rr` routine can be implemented in $O(b \log N)$ time. In the state-of-the-art ORAM constructions, such as Circuit ORAM [Wang et al. 2015], b can be parameterized as a constant (e.g., 4), which renders

the overall time complexity of rr to be $O(\log N)$, which is much faster than linear scanning over a trivial ORAM. Here is the code:

```
1  let rr (noram:noram R R') (idx:nat R) (tag:nat P):rnd R'*rnd R' =
2    let noram_cap = (length(noram) + 1) / 2 in
3    let bucket_size = length(noram[0]) in
4    let rec read_bucket bucket (idx:nat R) (i:nat P) (read:rnd R'*rnd R') =
5      if i=bucket_size then read
6      else
7        (* read out the current block *)
8        let isdummy, blk_idx, pos_tag, data = (bucket[i] ← (true, 0, 0, mk_dummy())) in
9        (* check if the current block is non-dummy and its index matches the queried one *)
10       let toswap:bit R = isdummy && blk_idx = idx in
11       let data, read = mux toswap read data in
12       let isdummy, _ = mux toswap false isdummy in
13       (* when toswap is false, this is equivalent to writting the data back; otherwise, read
14            stores the found block's data field, and is passed into the next iteration *)
15       let _ = (bucket[i] ← (isdummy, blk_idx, pos_tag, data)) in
16       read_bucket bucket idx (i+1) read in
17   let rec read_level (level:nat P) (read:nat R) =
18     let base:Nat P = (pow 2 level) − 1 in
19     (* compute the first index into the bucket array at depth level *)
20     if base >= length(noram) then read
21     else
22       let bucket_index:Nat P = base + (tag & base) in (* the bucket on the path to access *)
23       let bucket = noram[bucket_index] in
24       let read = readBucket bucket idx 0 read in
25       read_level (level + 1) read in
26   let read = mk_dummy () in
27   read_level 0 read
```

The implementation involves two nested loops: read_level (line 18-26) implements the outer loop, which iterates through all buckets on the path from the root to the leaf corresponding to tag; and read_bucket (line 4-17) iterates through all blocks in a bucket (this is essentially a Trivial ORAM read). Here are some things to notice. First, the NORAM itself is an array of arrays, and since array addresses are non-affine, we can read out noram[0] directly (line 3). On the other hand, buckets contain affine blocks (the data part), so we must replace a read block with a dummy one, as shown on lines 8 and 15. On line 10, toswap is computed to determine whether to swap out the block, i.e., whether its index matches the queried one. Line 10 **mux**es on toswap and picks either the block's data, or the argument read, which are each pairs of type rnd R' * rnd R'. While toswap is in region R, the pairs are in independent region R', so rule MuxFlip ensures the returned value is still in R'. If we were to store non-random data in the NORAM, we would require it to be in region R for this code to type check.

The add routine of NORAM is also simple: it adds the data into the bucket corresponding to the root node of the tree, which is the bucket at index 0 in the array. This is similar to the read_bucket routine above, so we omit its implementation. To avoid overflowing the root's bucket due to repeated adds, a tree-based ORAM employs an additional eviction routine to evict blocks in the buckets of NORAM nodes closer to the leaf buckets. This routine should also maintain the key invariant: each data block should always reside on the path corresponding to its position tag. Different tree-based ORAM implementations differ only in their choices of b and the eviction strategies. One simple eviction strategy picks random nodes at each level of the tree, reads a single non-empty block from

its bucket, and then writes that block one level further down either to the left or right; a dummy block is written in the opposite direction to make the operation oblivious.

6.2 Tree-based (recursive) ORAM

To use tree-based NORAM we need a way to get the position tag for reads and writes. A tree-based recursive ORAM employs an additional *position map* to solve this issue. Such a map PM maps any index $i \in 0..N$, where N is the capacity of ORAM, to randomly generated position tag, which will be given to the `rr` and `add` routines of NORAM. The Tree ORAM `tree_rr` routine will first access PM to get the position tag, and invoke NORAM's `rr` routine to retrieve the data block.

As mentioned in Section 3.2, one way to implement the position map is to just use a regular array stored in hidden memory, e.g., on-chip in a secure processor deployment of ORAM. However, this is not possible for MPC-based deployments, the adversary can observe the access pattern on the map itself. To avoid this problem, we can use another smaller Tree-based ORAM to store PM. In particular, the ORAM PM has N/c blocks, where each block contain c elements, where $c \geq 2$ is a parameter of the ORAM. Therefore, PM also contains another position map PM $'$, which can be stored as an ORAM of capacity N/c^2. Such a construction can continue recursively, until the position map at the bottom is small enough to be constructed as a Trivial ORAM. Therefore, there are $\log_c N$ NORAMs constructed to store those position maps, and thus the overall runtime of a recursive ORAM is $\log_c N$ times of the runtime of a NORAM. In the following, we implement the ORAM with $c = 2$.[3]

A full ORAM thus has the type `oram`, given below.

```
type oram R R' = ((noram R R') array) * trivial_oram R R'
type trivial_oram R R' = (to_block R R') array
type to_block R R' = bit * R * nat * (rnd R' * rnd R') where R ⫫ R'
```

In short, an ORAM is a sequence of NORAMs, where the first in the sequence contains the actual data, and the remainder serve as the position map, which eventually terminate with a trivial ORAM. A trivial ORAM is simply an array of blocks which contain the isdummy flag, the index, and the data (recall that trivial ORAMs themselves do not use position tags in their implementations). The data is a pair (i.e., $c = 2$) of position tags. The main idea is to think of the position map as essentially an array of size N but "stored" as a 2-D array: `array[N/2][2]`. In doing so, to access PM$[idx]$, we essentially access PM$[idx/2][idx \bmod 2]$.

We implement the `tree_rr` as a call to the function `tree_rr_rec`, which takes an additional public `level` argument, to indicate at which point in the list of NORAMs to start its work; it's initially called with level 0.

```
let rec tree_rr_rec (oram:oram R R') (idx:nat R) (level:nat P) : (rnd R' * rnd R') =
  let (norams,tr_oram) = oram in
  let levels:nat P = length(norams) in
  if (level >= levels) then ... (* do trivial ORAM case, returning block (rnd R' * rnd R') *)
  else
    let (r0,r1):rnd R' * rnd R' = tree_rr_rec oram (idx/2) (level+1) in
    let r0', tag = mux((idx mod 2) = 0) (rnd R' ()) r0 in
    let r1', tag = mux((idx mod 2) = 1) tag r1 in
    let _ = tree_add_rec oram (idx/2) (level+1) (r0', r1') in
    rr norams[level] idx (reveal tag)
let tree_rr (oram:oram R R') (idx:nat R) : (rnd R' * rnd R') = tree_rr_rec oram idx 0
```

[3]We present a readable, almost-correct version of the code first, and give the full picture in Section 6.3.

Line 4 checks whether we have hit the base case of the recursion, in which case we lookup idx in the tr_oram, returning back a rnd R' * rnd R' pair (code not shown). Otherwise, we enter the recursive case. Here, line 6 essentially reads out PM[$idx/2$], and lines 7 and 8 obliviously read out PM[$idx/2$][idx mod 2] into tag, replacing it with a freshly generated tag, to satisfy the affinity requirement. Finally, line 8 writes the updated block PM[$idx/2$] back (i.e., (r0', r1')), using an analogous tree_add_rec routine, for which a level can be specified. Finally, line 9 reveals the retrieved position tag for index idx, so that it can be passed into the rr routine of NORAM. Level 0 corresponds to the actual data of the ORAM, which is returned to the client.

The tree_add routine is similar so we do not show it all. As with tree_rr it recursively adds the corresponding bits of the position tag into the array of NORAMs. At each level of the recursion there is a snippet like the following:

```
1  let new_tag = rnd R' () in
2  let sec_tag = use new_tag in (* does NOT consume new_tag *)
3  let (r0,r1):rnd R' * rnd R' = tree_rr_rec oram (idx/2) (level+1) in
4  let r0', tag = mux ((idx mod 2) = 0) new_tag r0 in (* replaces with new tag *)
5  let r1', tag = mux ((idx mod 2) = 1) tag r1 in
6  let _ = tree_add_rec oram (idx/2) (level+1) (r0', r1') in
7  add norams[level] idx sec_tag data (* adds to Tree ORAM *)
```

Lines 1 and 2 generate a new tag, and make a secret copy of it. The new tag is then stored in the recursive ORAM—lines 3–5 are similar to tree_add_rec but replace the found tag with new_tag, not some garbage value, at the appropriate level of the position map (line 6). Finally, sec_tag is used to store the data in the appropriate level of the NORAM.

6.3 Limitations

The astute reader may have noticed that the code snippet for add will not type check. In particular, the sec_tag argument has type nat R' but add requires it to have type R. This is because the position tags for the NORAM at level are stored as the data of the NORAM at level+1, and these are in different regions. We cannot put them in the same region because we require a single NORAM's metadata and data to reside in different regions. We can solve this problem by having each level use the opposite pair of regions as the one above it. This solves the type error and does not compromise the NORAM independence requirement. Unfortunately, it means our oram type definition is made a bit more awkward, essentially it will be

```
1  type oram R R' = ((noram R R' * noram R' R) array) * trivial_oram R R'
```

Basically, tree_rr_rec will have to operate two levels at a time in order to satisfy the type checker, but the basic logic will be the same. In addition, tree_rr_rec and tree_rr require the type of idx to be nat S, i.e., probability region $\bot$ rather than R. Because $\bot \sqsubseteq$ R and $\bot \sqsubseteq$ R' in the region lattice, idx can be passed to rr at both of the unrolled levels, one of which requires it to be R and the other, R'.

A more unfortunate limitation is that we cannot implement an NORAM on which we can also implement an oblivious stack. To see why, consider the rr operation implemented above, but with the data having type nat R * rnd R', as required for an oblivious stack (the first component is the stack element's value, and the second is the position tag of the stack's next element). The issue is that the stack element's position tag is from the same region as the position tags in the underlying NORAM, which violates the independence requirement of the **mux** on line 11 of rr. However, this violation is a false positive: While the tags happen to be in the same region, they are actually independent—no tag's generation is conditioned on any other's value, and the same tag is never stored in the same block. Indeed, using the more general NORAM signature presented in Section 3.2, $L_{\textbf{obliv}}$'s type system confirms that the stack in Figure 3 is secure. Checking the implementation of

the stack and the NORAM end-to-end likely requires a more fine-grained, generative treatment of regions and the use of existential types. Such features may simplify the solution to recursion problem as well. We are pursuing them in ongoing work.

7 RELATED WORK

Lampson first pointed out various covert, or "side," channels of information leakage during a program's execution [Lampson 1973]. Defending against side-channel leakage is challenging. Previous works have attempted to thwart such leakage from various angles:

- Processor architectures that mitigate leakage through timing [Kocher et al. 2004; Liu et al. 2012], power consumption [Kocher et al. 2004], or memory-traces [Fletcher et al. 2014; Liu et al. 2015; Maas et al. 2013; Ren et al. 2013].
- Program analysis techniques that formally ensure that a program has bounded or no leakage through instruction traces [Molnar et al. 2006], timing channels [Agat 2000; Molnar et al. 2006; Russo et al. 2006; Zhang et al. 2012, 2015], or memory traces [Liu et al. 2015, 2013, 2014].
- Algorithmic techniques that transform programs and algorithms to their side-channel-mitigating or side-channel-free counterparts while introducing only mild costs – e.g., works on mitigating timing channel leakage [Askarov et al. 2010; Barthe et al. 2010; Zhang et al. 2011], and on preventing memory-trace leakage [Blanton et al. 2013; Eppstein et al. 2010; Goldreich 1987; Goldreich and Ostrovsky 1996; Goodrich et al. 2012; Shi et al. 2011; Stefanov et al. 2013; Wang et al. 2015, 2014; Zahur and Evans 2013].

Often, the most effective and efficient is through a comprehensive co-design approach combining these areas of advances – in fact, several aforementioned works indeed combine (a subset of) algorithms, architecture, and programming language techniques [Fletcher et al. 2014; Liu et al. 2015; Ren et al. 2013; Zhang et al. 2012, 2015].

Our core language not only generalizes a line of prior works on timing channel security [Agat 2000], program counter security [Molnar et al. 2006], and memory-trace obliviousness [Liu et al. 2015, 2013, 2014], but also provides the first distinct core language that captures the essence of memory-trace obliviousness without treating key mechanisms—notably, the use of randomness—as a black box. Thus we can express state-of-the-art algorithmic results and formally reason about the security of their implementations, thus building a bridge between algorithmic and programming language techniques.

ObliVM [Liu et al. 2015] is a language for programming oblivious algorithms intended to be run as secure multiparty computations [Yao 1986]. Its type system also employs affine types to ensure random numbers are used at most once. However, there it provides no mechanism to disallow constructing a non-uniformly distributed random number. When such random numbers are generated, they can be distinguished by an attacker from uniformly distributed random numbers when being revealed. Therefore, the type system in ObliVM does not guarantee obliviousness. $L_{\mathbf{obliv}}$'s use of probability regions enforces that all random numbers are uniformly random, and thus eliminates this channel of information leakage. Moreover, we prove that this mechanism (and the others in $L_{\mathbf{obliv}}$) are sufficient to prove PMTO.

Our work belongs to a large category of work that aims to statically enforce *noninterference*, e.g., by typing [Sabelfeld and Myers 2006; Volpano et al. 1996]. Our probabilistic memory trace obliviousness property bears some resemblance to probabilistic notions of noninterference. Much prior work [Ngo et al. 2014; Russo and Sabelfeld 2006; Sabelfeld and Sands 2000; Smith 2003] is concerned with how random choices made by a thread scheduler could cause the distribution of visible events to differ due to the values of secrets. Here, the source of nondeterminism is the (external) scheduler, rather than the program itself, as in our case. Smith and Alpízar [2006, 2007]

consider how the influence of random numbers may affect the likelihood of certain outcomes, mostly being concerned with termination channels. Their programming model is not as rich as ours, as a secret random number is never permitted to be made public; such an ability is the main source of complexity in $L_{\mathbf{obliv}}$, and is crucial for supporting oblivious algorithms.

Some prior work aims to quantify the information released by a (possibly randomized) program (e.g., Köpf and Rybalchenko [2013]; Mu and Clark [2009]) according to entropy-based measures. Work on verifying the correctness of differentially private algorithms [Barthe et al. 2013; Zhang and Kifer 2017] essentially aims to bound possible leakage; by contrast, we enforce that *no* information leaks due to a program's execution.

8 CONCLUSIONS

This paper has presented $L_{\mathbf{obliv}}$, a core language suitable for expressing computations whose execution should be oblivious to a powerful adversary who can observe an execution's trace of instructions and memory accesses, but not see private values. Unlike prior formalisms, $L_{\mathbf{obliv}}$ can be used to express probabilistic algorithms whose security depends crucially on the use of randomness. To do so, $L_{\mathbf{obliv}}$ tracks the use of randomly generated numbers via a substructural (affine) type system, and employs a novel concept called *probability regions*. The latter are used to track a random number's probabilistic (in)dependence on other random numbers. We have proved that together these mechanisms ensure that a random number's revelation in the visible trace does not perturb the distribution of possible events so as to make secrets more likely. We have demonstrated that $L_{\mathbf{obliv}}$'s type system is powerful enough to accept sophisticated algorithms, including forms of oblivious RAMs and oblivious data structures. Such data structures were out of the reach of prior type systems. As such, our proof-of-concept implementations represent the first automated proofs that these algorithms are secure.

We are currently pursuing two threads of ongoing work. First, we are working on a more powerful notion of probability region that is finer grained so as to accept oblivious data structure algorithms, end to end. Second, we are working to expand the general expressiveness of the language. For example, to support containers requires some form of parametric polymorphism, and to support some deployments of ORAM requires bit-width polymorphism, implementable with dependent types. Ultimately we hope to integrate our ideas into ObliVM [Liu et al. 2015], a state-of-the-art compiler for oblivious algorithms, and thereby ensure that well-typed programs are indeed secure.

ACKNOWLEDGMENTS

We thank Aseem Rastogi and Kesha Heitala for comments on earlier drafts of this paper, and Elaine Shi for helpful suggestions and comments on the work while it was underway. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1563722, CNS-1314857, and CNS-1111599, and by DARPA under contracts FA8750-15-2-0104 and FA8750-16-C-0022. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Johan Agat. 2000. Transforming out Timing Leaks. In *POPL*.

Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive black-box mitigation of timing channels. In *CCS*.

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3 (2013), 9:1–9:49.

Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. 2010. Security of multithreaded programs by compilation. *ACM Transactions on Information and System Security (TISSEC)* 13, 3 (2010), 21.

Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ASIA CCS*.

David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *USENIX Security*.

D. Dolev and A. C. Yao. 1981. On the Security of Public Key Protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (SFCS)*.

David Eppstein, Michael T. Goodrich, and Roberto Tamassia. 2010. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*.

Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*.

J.A. Goguen and J. Meseguer. 1982. Security policy and security models. In *IEEE S & P*.

O. Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*.

O. Goldreich, S. Micali, and A. Wigderson. 1987. How to play ANY mental game. In *STOC*.

Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* (1996).

Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. 2012. Data-Oblivious Graph Drawing Model and Algorithms. *CoRR* abs/1209.0756 (2012).

Matt Hoekstra. 2015. Intel SGX for Dummies (Intel SGX Design Objectives). https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx. (2015).

Mohammad Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Network and Distributed System Security Symposium (NDSS)*.

Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. 2004. Security As a New Dimension in Embedded System Design. In *Proceedings of the 41st Annual Design Automation Conference (DAC '04)*. 753–760. Moderator-Ravi, Srivaths.

Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.

Boris Köpf and Andrey Rybalchenko. 2013. Automation of quantitative information-flow analysis. In *Formal Methods for Dynamical Systems*.

Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* (1973).

Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS*.

Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *CSF*.

Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. 2014. Automating Efficient RAM-Model Secure Computation. In *IEEE S & P*.

Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *IEEE S & P*.

Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. 2012. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *ICCD*.

Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. 2013. Phantom: Practical Oblivious Computation in a Secure Processor. In *CCS*.

David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The Program Counter Security Model: Automatic Detection and Removal of Control-flow Side Channel Attacks. In *ICISC*.

Chunyan Mu and David Clark. 2009. An abstraction quantifying information flow over probabilistic semantics. In *Workshop on Quantitative Aspects of Programming Languages (QAPL)*.

Tri Minh Ngo, Mariëlle Stoelinga, and Marieke Huisman. 2014. Effective verification of confidentiality for multi-threaded programs. *Journal of computer security* 22, 2 (2014).

Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. 2013. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*.

Alejandro Russo, John Hughes, David A. Naumann, and Andrei Sabelfeld. 2006. Closing Internal Timing Channels by Transformation. In *Annual Asian Computing Science Conference (ASIAN)*.

Alejandro Russo and Andrei Sabelfeld. 2006. Securing interaction between threads and the scheduler. In *CSF-W*.

A. Sabelfeld and A. C. Myers. 2006. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (Sept. 2006).

Andrei Sabelfeld and David Sands. 2000. Probabilistic noninterference for multi-threaded programs. In *CSF-W*.

Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT*.

Geoffrey Smith. 2003. Probabilistic noninterference through weak probabilistic bisimulation. In *CSF-W*.

Geoffrey Smith and Rafael Alpízar. 2006. Secure Information Flow with Random Assignment and Encryption. In *Workshop on Formal Methods in Security (FMSE)*.

Geoffrey Smith and Rafael Alpízar. 2007. Fast Probabilistic Simulation, Nontermination, and Secure Information Flow. In *PLAS*.

Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM – an Extremely Simple Oblivious RAM Protocol. In *CCS*.

G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS*.

David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *SIGOPS Oper. Syst. Rev.* 34, 5 (Nov. 2000).

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2-3 (Jan. 1996).

Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS*.

Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *CCS*.

Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *FOCS*.

Samee Zahur and David Evans. 2013. Circuit Structures for Improving Efficiency of Security and Privacy Tools. In *S & P*.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2011. Predictive Mitigation of Timing Channels in Interactive Systems. In *CCS*.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-based Control and Mitigation of Timing Channels. In *PLDI*.

Danfeng Zhang and Daniel Kifer. 2017. LightDP: Towards Automating Differential Privacy Proofs. In *POPL*.

Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*.

Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2004. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. *SIGARCH Comput. Archit. News* 32, 5 (Oct. 2004).

A APPENDIX

$x \in X$		Notation for elements x of a parameterized set X
$\vec{x}^N \in X^N$		Sequence of elements X of length N
$\vec{x}^N \in X^*$		Sequence of elements X of *some* length N
$\tilde{x}^R \in \widetilde{X}^R$	$:= \mathbb{B}^R \to X$	Distributions of elements X over R coin flips
$i \in idx^N$	$:= \{1, .., N\}$	Index into sequences of length N
$\theta \in dep$	$:= \wp(\mathbb{N})$	Probabilistic Dependency Set

$$\mathfrak{c}[\cdot = \cdot] : \forall R\ X.\ \widetilde{X}^R \times X \to \mathbb{N} \qquad \mathfrak{c}\left[\tilde{x} = x\right] := \left\| \left\{ \vec{b} \;\middle|\; \tilde{x}\left(\vec{b}\right) = x \right\} \right\|$$
$$\mathfrak{p}[\cdot = \cdot] : \forall R\ X.\ \widetilde{X}^R \times X \to \mathbb{Q} \qquad \mathfrak{p}\left[\tilde{x} = x\right] := \frac{\mathfrak{c}\left[\tilde{x} = x\right]}{2^R}$$

$$\mathfrak{c}[\cdot = \cdot \wedge \cdot = \cdot] : \forall R\ X_1\ X_2.\ \widetilde{X}_1^R \times X_1 \times \widetilde{X}_2^R \times X_2 \to \mathbb{N}$$
$$\mathfrak{p}[\cdot = \cdot \wedge \cdot = \cdot] : \forall R\ X_1\ X_2.\ \widetilde{X}_1^R \times X_1 \times \widetilde{X}_2^R \times X_2 \to \mathbb{Q}$$
$$\mathfrak{p}[\cdot = \cdot \mid \cdot = \cdot] : \forall R\ X_1\ X_2.\ \widetilde{X}_1^R \times X_1 \times \widetilde{X}_2^R \times X_2 \to \mathbb{Q}$$

$$\mathfrak{c}\left[\tilde{x}_1 = x_1 \wedge \tilde{x}_2 = x_2\right] := \left\| \left\{ \vec{b} \;\middle|\; \tilde{x}_1\left(\vec{b}\right) = x_1 \wedge \tilde{x}_2\left(\vec{b}\right) = x_2 \right\} \right\|$$
$$\mathfrak{p}\left[\tilde{x}_1 = x_1 \wedge \tilde{x}_2 = x_2\right] := \frac{\mathfrak{c}\left[\tilde{x}_1 = x_1 \wedge \tilde{x}_2 = x_2\right]}{2^R}$$
$$\mathfrak{p}\left[\tilde{x}_1 = x_1 \mid \tilde{x}_2 = x_2\right] := \frac{\mathfrak{c}\left[\tilde{x}_1 = x_1 \wedge \tilde{x}_2 = x_2\right]}{\mathfrak{c}\left[\tilde{x}_2 = x_2\right]}$$

$$\cdot \mathbin{\widetilde{\odot}} \cdot : \forall R.\ \widetilde{\mathbb{N}}^R \times \widetilde{\mathbb{N}}^R \to \widetilde{\mathbb{N}}^R$$
$$\widetilde{cond} : \forall R\ X.\ \widetilde{\mathbb{B}}^R \times \widetilde{X}^R \times \widetilde{X}^R \to \widetilde{X}^R$$
$$\widetilde{mux} : \forall R\ X.\ \widetilde{\mathbb{B}}^R \times \widetilde{X}^R \times \widetilde{X}^R \to \widetilde{X}^R \times \widetilde{X}^R$$

$$\tilde{n}_1 \mathbin{\widetilde{\odot}} \tilde{n}_2 := \lambda\vec{b}.\ [\![\odot]\!]\left(\tilde{n}_1\left(\vec{b}\right), \tilde{n}_2\left(\vec{b}\right)\right)$$
$$\widetilde{cond}\left(\tilde{x}_1, \tilde{x}_2, \tilde{x}_3\right) := \lambda\vec{b}.\ \begin{cases} \tilde{x}_2\left(\vec{b}\right) & \text{if } \tilde{x}_1\left(\vec{b}\right) = \mathtt{O} \\ \tilde{x}_3\left(\vec{b}\right) & \text{if } \tilde{x}_1\left(\vec{b}\right) = \mathtt{I} \end{cases}$$
$$\widetilde{mux}\left(\tilde{x}_1, \tilde{x}_2, \tilde{x}_3\right) := \left\langle \widetilde{cond}\left(\tilde{x}_1, \tilde{x}_2, \tilde{x}_3\right), \widetilde{cond}\left(\tilde{x}_1, \tilde{x}_3, \tilde{x}_2\right) \right\rangle$$

$$\lfloor\cdot\rfloor : \forall R\ X.\ X \to \widetilde{X}^R \qquad \lfloor x \rfloor := \lambda\_.\ x$$
$$sel[\cdot] : \forall R\ X.\ idx^R \to \widetilde{X}^R \qquad sel[i] := \lambda\vec{b}^R.\ b_i$$
$$\tilde{\uparrow} : \forall R\ R'\ X.\ \widetilde{X}^R \to \widetilde{X}^{R+R'} \qquad \tilde{\uparrow}\tilde{x}^R := \lambda\vec{b}^{R+R'}.\ \tilde{x}^R(b_1 \ldots b_R)$$

$$\tilde{x}_1 \approx \tilde{x}_2 := \forall x.\ \mathfrak{p}\left[\tilde{x}_1 = x\right] = \mathfrak{p}\left[\tilde{x}_2 = x\right]$$
$$\tilde{x}_1 \approx_{/\!/\theta} \tilde{x}_2 := \forall x.\ \mathfrak{p}\left[\tilde{x}_1 = x \mid \theta\right] = \mathfrak{p}\left[\tilde{x}_2 = x \mid \theta\right]$$
$$stable\left[\tilde{x} \mathbin{/\!/} \theta\right] := \forall x_1\ x_2.\ \mathfrak{p}\left[\tilde{x} = x_1 \mid \theta\right] = \mathfrak{p}\left[\tilde{x} = x_2 \mid \theta\right]$$
$$\tilde{x}_1 \perp\!\!\!\perp \tilde{x}_2 := \forall x_1\ x_2.\ \mathfrak{p}\left[\tilde{x}_1 = x_1 \wedge \tilde{x}_2 = x_2\right] = \mathfrak{p}\left[\tilde{x}_1 = x_1\right]\mathfrak{p}\left[\tilde{x}_2 = x_2\right]$$
$$\tilde{x} \propto \theta := \forall i \notin \theta.\ \tilde{x} \perp\!\!\!\perp sel[i]$$

Fig. 8. Distributions

$$\alpha \in addr := \{0, 1, \ldots\} \qquad \widehat{\gamma}^R \in \widehat{env}^R := var \rightharpoonup \widehat{\mathcal{V}}^R \uplus \{\bullet\}$$
$$\widehat{v}^R \in \widehat{\mathcal{V}}^R ::= \tilde{\iota}_\ell^R \mid \tilde{\alpha}^R \mid \left\langle \widehat{v}^R, \widehat{v}^R \right\rangle \qquad \widehat{\sigma}^R \in \widehat{env}^R := addr \rightharpoonup \left(\widehat{\mathcal{V}}^R\right)^*$$

$$\widehat{\uparrow} : \forall R\ R'.\ \widehat{\mathcal{V}}^R \to \widehat{\mathcal{V}}^{R+R'} \qquad \begin{aligned} \widehat{\uparrow}(\tilde{\iota}) &:= \tilde{\uparrow}(\tilde{\iota}) \\ \widehat{\uparrow}(\tilde{\alpha}) &:= \tilde{\uparrow}(\tilde{\alpha}) \\ \widehat{\uparrow}(\langle \widehat{v}_1, \widehat{v}_2 \rangle) &:= \left\langle \widehat{\uparrow}\widehat{v}_1, \widehat{\uparrow}\widehat{v}_2 \right\rangle \end{aligned}$$

$$\overset{env}{\widehat{\uparrow}} : \forall R\ R'.\ \widehat{env}^R \to \widehat{env}^{R+R'} \qquad \left(\overset{env}{\widehat{\uparrow}}\widehat{\gamma}\right)(x) := \widehat{\uparrow}\widehat{\gamma}(x)$$
$$\overset{store}{\widehat{\uparrow}} : \forall R\ R'.\ \widehat{store}^R \to \widehat{store}^{R+R'} \qquad \left(\overset{store}{\widehat{\uparrow}}\widehat{\sigma}\right)(\alpha)_n := \widehat{\uparrow}\widehat{\sigma}(\alpha)_n$$

$$\widehat{cond} : \forall R.\ \widehat{\mathcal{V}}^R \times \widehat{\mathcal{V}}^R \times \widehat{\mathcal{V}}^R \rightharpoonup \widehat{\mathcal{V}}^R$$
$$\widehat{mux} : \forall R.\ \widehat{\mathcal{V}}^R \times \widehat{\mathcal{V}}^R \times \widehat{\mathcal{V}}^R \rightharpoonup \widehat{\mathcal{V}}^R \times \widehat{\mathcal{V}}^R$$

$$\begin{aligned} \widehat{cond}\left(\tilde{b}_\ell, \tilde{\iota}_1, \tilde{\iota}_2\right) &:= \widetilde{cond}\left(\tilde{b}, \tilde{\iota}_1, \tilde{\iota}_2\right) \\ \widehat{cond}\left(\tilde{b}_\ell, \tilde{\alpha}_1, \tilde{\alpha}_2\right) &:= \widetilde{cond}\left(\tilde{b}, \tilde{\alpha}_1, \tilde{\alpha}_2\right) \\ \widehat{cond}\left(\tilde{b}_\ell, \langle \widehat{v}_{11}, \widehat{v}_{21} \rangle, \langle \widehat{v}_{12}, \widehat{v}_{22} \rangle\right) &:= \left\langle \widehat{cond}\left(\tilde{b}_\ell, \widehat{v}_{11}, \widehat{v}_{12}\right), \widehat{cond}\left(\tilde{b}_\ell, \widehat{v}_{21}, \widehat{v}_{22}\right) \right\rangle \\ \widehat{mux}\left(\tilde{b}_\ell, \widehat{v}_1, \widehat{v}_2\right) &:= \left\langle \widehat{cond}\left(\tilde{b}_\ell, \widehat{v}_1, \widehat{v}_2\right), \widehat{cond}\left(\tilde{b}_\ell, \widehat{v}_2, \widehat{v}_1\right) \right\rangle \end{aligned}$$

$$join : \forall R.\ \widetilde{\widehat{\mathcal{V}}^R}^R \rightharpoonup \widehat{\mathcal{V}}^R \qquad \begin{aligned} join\left(\tilde{\tilde{\iota}}\right) &:= \lambda \vec{b}.\ \tilde{\tilde{\iota}}\left(\vec{b}\right)\left(\vec{b}\right) \\ join\left(\tilde{\tilde{\alpha}}\right) &:= \lambda \vec{b}.\ \tilde{\tilde{\alpha}}\left(\vec{b}\right)\left(\vec{b}\right) \\ join\left(\widetilde{\langle \widehat{v}_1, \widehat{v}_2 \rangle}\right) &:= \left\langle join\left(\tilde{\widehat{v}}_1\right), join\left(\tilde{\widehat{v}}_2\right) \right\rangle \end{aligned}$$

$$read : \forall R.\ \widetilde{addr}^R \times \widetilde{\mathbb{N}}^R \times \widetilde{store}^R \rightharpoonup \widehat{\mathcal{V}}^R$$
$$read{+}write : \forall R.\ \widetilde{addr}^R \times \widetilde{\mathbb{N}}^R \times \widehat{\mathcal{V}}^R \times \widetilde{store}^R \rightharpoonup \widehat{\mathcal{V}}^R \times \widetilde{store}^R$$

$$read\left(\tilde{\alpha}, \tilde{n}_\ell, \widehat{\sigma}\right) := join\left(\lambda \vec{b}.\ \widehat{\sigma}\left(\tilde{\alpha}\left(\vec{b}\right)\right)\left[\tilde{n}\left(\vec{b}\right)\right]\right)$$
$$read{+}write\left(\tilde{\alpha}, \tilde{n}_\ell, \widehat{v}, \widehat{\sigma}\right) := \left\langle read\left(\tilde{\alpha}, \tilde{n}, \widehat{\sigma}\right), \widehat{\sigma}' \right\rangle$$
$$\text{where } \widehat{\sigma}'(\alpha)[n] := join\left(\lambda \vec{b}.\ \begin{cases} \widehat{v} & \text{if } \tilde{\alpha}\left(\vec{b}\right) = \alpha \text{ and } \tilde{n}\left(\vec{b}\right) = n \\ \widehat{\sigma}(\alpha)[n] & \text{if } \tilde{\alpha}\left(\vec{b}\right) \neq \alpha \text{ or } \tilde{n}\left(\vec{b}\right) \neq n \end{cases}\right)$$

Fig. 9. Domains

$$\overset{pico}{\mathcal{E}}[\![\cdot]\!] : pico \to \forall R.\ \widehat{env}^R \rightharpoonup \widehat{\mathcal{V}}^R \times \widehat{env}^R$$
$$\overset{atom}{\mathcal{E}}[\![\cdot]\!] : atom \to \forall R.\ \widehat{env}^R \times \widehat{store}^R \rightharpoonup \exists R'.\ \widehat{\mathcal{V}}^{R'} \times \widehat{env}^{R'} \times \widehat{store}^{R'}$$
$$\overset{exp}{\mathcal{E}}[\![\cdot]\!] : expr \to \forall R.\ \widehat{env}^R \times \widehat{store}^R \rightharpoonup \exists R'.\ \widehat{\mathcal{V}}^{R'} \times \widehat{env}^{R'} \times \widehat{store}^{R'}$$

$$\overset{pico}{\mathcal{E}}[\![x]\!]^{\cup}(\widehat{\gamma}) := \langle \widehat{\gamma}(x), \widehat{\gamma} \rangle \qquad \overset{pico}{\mathcal{E}}[\![x]\!]^{\sqcup}(\widehat{\gamma}) := \langle \widehat{\gamma}(x), \widehat{\gamma}[x \mapsto \bullet] \rangle \qquad \overset{pico}{\mathcal{E}}[\![\iota_\ell]\!](\widehat{\gamma}) := \langle \lfloor \iota_\ell \rfloor, \widehat{\gamma} \rangle$$

$$\overset{atom}{\mathcal{E}}[\![p]\!](\widehat{\gamma}, \widehat{\sigma}) := \langle \widehat{v}, \widehat{\gamma}', \widehat{\sigma} \rangle \quad \text{where} \quad \langle \widehat{v}, \widehat{\gamma}' \rangle := \overset{pico}{\mathcal{E}}[\![p]\!](\widehat{\gamma})$$
$$\overset{atom}{\mathcal{E}}[\![p^\circ]\!](\widehat{\gamma}, \widehat{\sigma}) := \left\langle \widetilde{cond}\left(\tilde{b}, \lfloor 0 \rfloor, \lfloor 1 \rfloor\right), \widehat{\gamma}', \widehat{\sigma} \right\rangle \quad \text{where} \quad \left\langle \tilde{b}, \widehat{\gamma}' \right\rangle := \overset{pico}{\mathcal{E}}[\![p]\!](\widehat{\gamma})$$
$$\overset{atom}{\mathcal{E}}[\![p_1 \odot p_2]\!](\widehat{\gamma}, \widehat{\sigma}) := \left\langle \tilde{n}_1 \mathbin{\widetilde{\odot}} \tilde{n}_2, \widehat{\gamma}'', \widehat{\sigma} \right\rangle$$
$$\text{where} \quad \langle \tilde{n}_1, \widehat{\gamma}' \rangle := \overset{pico}{\mathcal{E}}[\![p_1]\!](\widehat{\gamma}) \quad \langle \tilde{n}_2, \widehat{\gamma}'' \rangle := \overset{pico}{\mathcal{E}}[\![p_2]\!](\widehat{\gamma}')$$
$$\overset{atom}{\mathcal{E}}[\![\texttt{asnat}(p)]\!](\widehat{\gamma}, \widehat{\sigma}) := \left\langle \widetilde{asnat}\left(\tilde{b}\right), \widehat{\gamma}', \widehat{\sigma} \right\rangle \quad \text{where} \quad \left\langle \tilde{b}, \widehat{\gamma}' \right\rangle := \overset{pico}{\mathcal{E}}[\![p]\!](\widehat{\gamma})$$
$$\overset{atom}{\mathcal{E}}[\![\texttt{asbit}(p)]\!](\widehat{\gamma}, \widehat{\sigma}) := \left\langle \widetilde{asbit}\left(\tilde{n}\right), \widehat{\gamma}', \widehat{\sigma} \right\rangle \quad \text{where} \quad \langle \tilde{n}, \widehat{\gamma}' \rangle := \overset{pico}{\mathcal{E}}[\![p]\!](\widehat{\gamma})$$
$$\overset{atom}{\mathcal{E}}[\![\texttt{flip}^{\rho}_{\ell}]\!]\left(\widehat{\gamma}^R, \widehat{\sigma}^R\right) := \left\langle sel[R+1], \overset{env}{\widehat{\uparrow}} \widehat{\gamma}^R, \overset{store}{\widehat{\uparrow}} \widehat{\sigma}^R \right\rangle$$
$$\overset{atom}{\mathcal{E}}[\![\texttt{use}(x)]\!](\widehat{\gamma}, \widehat{\sigma}) := \langle \widehat{\gamma}(x), \widehat{\gamma}, \widehat{\sigma} \rangle$$
$$\overset{atom}{\mathcal{E}}[\![\texttt{reveal}(x)]\!](\widehat{\gamma}, \widehat{\sigma}) := \langle \widehat{\gamma}(x), \widehat{\gamma}[x \mapsto \bullet], \widehat{\sigma} \rangle$$
$$\overset{atom}{\mathcal{E}}[\![\texttt{mux}(p_1, p_2, p_3)]\!](\widehat{\gamma}, \widehat{\sigma}) := \langle \widetilde{mux}\left(\widehat{v}_1, \widehat{v}_2, \widehat{v}_3\right), \widehat{\gamma}''', \widehat{\sigma} \rangle$$
$$\text{where} \quad \langle \widehat{v}_1, \widehat{\gamma}' \rangle := \overset{pico}{\mathcal{E}}[\![p_1]\!](\widehat{\gamma}) \quad \langle \widehat{v}_2, \widehat{\gamma}'' \rangle := \overset{pico}{\mathcal{E}}[\![p_2]\!](\widehat{\gamma}') \quad \langle \widehat{v}_2, \widehat{\gamma}''' \rangle := \overset{pico}{\mathcal{E}}[\![p_2]\!](\widehat{\gamma}'')$$
$$\overset{atom}{\mathcal{E}}[\![\langle p_1, p_2 \rangle]\!](\widehat{\gamma}, \widehat{\sigma}) := \langle \langle \widehat{v}_1, \widehat{v}_2 \rangle, \widehat{\gamma}'', \widehat{\sigma} \rangle$$
$$\text{where} \quad \langle \widehat{v}_1, \widehat{\gamma}' \rangle := \overset{pico}{\mathcal{E}}[\![p_1]\!](\widehat{\gamma}) \quad \langle \widehat{v}_2, \widehat{\gamma}'' \rangle := \overset{pico}{\mathcal{E}}[\![p_2]\!](\widehat{\gamma}')$$
$$\text{where} \quad \langle \tilde{n}_1, \widehat{\gamma}' \rangle := \overset{pico}{\mathcal{E}}[\![p_1]\!](\widehat{\gamma}) \quad \langle \tilde{n}_2, \widehat{\gamma}'' \rangle := \overset{pico}{\mathcal{E}}[\![p_2]\!](\widehat{\gamma}')$$

Fig. 10. Denotational Evaluation Semantics (1)

$$\overset{atom}{\mathcal{E}}[\![\texttt{array}^N(p_1,\ldots,p_N)]\!]\,(\widehat{\gamma}_0,\widehat{\sigma}) := \langle \lfloor\alpha\rfloor, \widehat{\gamma}_n, \widehat{\sigma}\,[\alpha \mapsto [n \mapsto \widehat{v}_n]]\rangle$$
$$\textit{where}\quad \langle \widehat{v}_n, \widehat{\gamma}_n, \rangle := \overset{pico}{\mathcal{E}}[\![p_n]\!]\,(\widehat{\gamma}_{n-1}) \quad \alpha := \textit{fresh}\,(\widehat{\sigma})$$
$$\overset{atom}{\mathcal{E}}[\![p_1[p_2]]\!]\,(\widehat{\gamma},\widehat{\sigma}) := \langle \textit{read}\,(\widehat{v}_1,\widehat{v}_2,\widehat{\sigma}), \widehat{\gamma}'', \widehat{\sigma}\rangle$$
$$\textit{where}\quad \langle \widehat{v}_1,\widehat{\gamma}'\rangle := \overset{pico}{\mathcal{E}}[\![p_1]\!]\,(\widehat{\gamma}) \quad \langle \widehat{v}_2,\widehat{\gamma}''\rangle := \overset{pico}{\mathcal{E}}[\![p_2]\!]\,(\widehat{\gamma}')$$
$$\overset{atom}{\mathcal{E}}[\![p_1[p_2] \leftarrow p_3]\!]\,(\widehat{\gamma},\widehat{\sigma}) := \langle \widehat{v}, \widehat{\gamma}''', \widehat{\sigma}'\rangle$$
$$\textit{where}\quad \langle \widehat{v}_1,\widehat{\gamma}'\rangle := \overset{pico}{\mathcal{E}}[\![p_1]\!]\,(\widehat{\gamma}) \quad \langle \widehat{v}_2,\widehat{\gamma}''\rangle := \overset{pico}{\mathcal{E}}[\![p_2]\!]\,(\widehat{\gamma}') \quad \langle \widehat{v}_3,\widehat{\gamma}'''\rangle := \overset{pico}{\mathcal{E}}[\![p_2]\!]\,(\widehat{\gamma}'')$$
$$\langle \widehat{v}, \widehat{\sigma}'\rangle := \textit{read+write}\,(\widehat{v}_1,\widehat{v}_2,\widehat{v}_3,\widehat{\sigma})$$

$$\overset{exp}{\mathcal{E}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma}) := \overset{atom}{\mathcal{E}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma})$$
$$\overset{exp}{\mathcal{E}}[\![\texttt{let } x := a \texttt{ in } e]\!]\,(\widehat{\gamma},\widehat{\sigma}) := \overset{exp}{\mathcal{E}}[\![e]\!]\,(\widehat{\gamma}'[x \mapsto \widehat{v}], \widehat{\sigma}')$$
$$\textit{where}\quad \langle \widehat{v}, \widehat{\gamma}', \widehat{\sigma}'\rangle := \overset{atom}{\mathcal{E}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma})$$
$$\overset{exp}{\mathcal{E}}[\![\texttt{let } x_1, x_2 := a \texttt{ in } e]\!]\,(\widehat{\gamma},\widehat{\sigma}) := \overset{exp}{\mathcal{E}}[\![e]\!]\,(\widehat{\gamma}'\,[x_1 \mapsto \widehat{v}_1, x_2 \mapsto \widehat{v}_2]\,, \widehat{\sigma}')$$
$$\textit{where}\quad \langle\langle \widehat{v}_1, \widehat{v}_2\rangle, \widehat{\gamma}', \widehat{\sigma}'\rangle := \overset{atom}{\mathcal{E}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma})$$
$$\overset{exp}{\mathcal{E}}[\![\texttt{if } p \texttt{ then } e_1 \texttt{ else } e_2]\!]\,(\widehat{\gamma},\widehat{\sigma}) := \langle \widehat{v}, \widehat{\gamma}'', \widehat{\sigma}'\rangle$$
$$\textit{where}\quad \left\langle \widehat{v}^R, \widehat{\gamma'}^R \right\rangle := \overset{pico}{\mathcal{E}}[\![p]\!]\,(\widehat{\gamma}) \qquad \widehat{v}^{R_1 \sqcup R_2} := \widehat{\textit{cond}}\left(\widehat{\uparrow} v, \widehat{\uparrow} v_1, \widehat{\uparrow} v_2\right)$$
$$\left\langle \widehat{v}_1^{R_1}, \widehat{\gamma}_1^{R_1}, \widehat{\sigma}_1^{R_1} \right\rangle := \overset{exp}{\mathcal{E}}[\![e_1]\!]\,(\widehat{\gamma}', \widehat{\sigma}) \qquad \widehat{\gamma''}^{R_1 \sqcup R_2}(x) := \widehat{\textit{cond}}\left(\widehat{\uparrow} v, \widehat{\uparrow} \gamma_1(x), \widehat{\uparrow} \gamma_2(x)\right)$$
$$\left\langle \widehat{v}_2^{R_2}, \widehat{\gamma}_2^{R_2}, \widehat{\sigma}_2^{R_2} \right\rangle := \overset{exp}{\mathcal{E}}[\![e_2]\!]\,(\widehat{\gamma}', \widehat{\sigma}) \qquad \widehat{\sigma'}^{R_1 \sqcup R_2}(\alpha) := \widehat{\textit{cond}}\left(\widehat{\uparrow} v, \widehat{\uparrow} \sigma_1(\alpha), \widehat{\uparrow} \sigma_2(\alpha)\right)$$

Fig. 11. Denotational Evaluation Semantics (2)

$$l \in loc ::= \ldots \textit{source code locations} \ldots \qquad \gamma \in env := var \rightharpoonup \mathcal{V}$$
$$v \in \mathcal{V} ::= \iota_\ell \mid \alpha \mid \langle v, v \rangle \qquad \sigma \in store := addr \rightharpoonup \mathcal{V}^*$$
$$t \in trace := (loc \times \mathcal{V} \times env \times store)^*$$

$$\cdot \otimes \cdot : \forall R\ X_1\ X_2.\ \widetilde{X_1}^R \times \widetilde{X_2}^R \to \widetilde{X_1 \times X_2}^R$$
$$\cdot \mathbin{\widetilde{+}} \cdot : \forall R\ X.\ \widetilde{X^*}^R \times \widetilde{X^*}^R \to \widetilde{X^*}$$

$$\tilde{x}_1 \otimes \tilde{x}_2 := \lambda \vec{b}.\ \left\langle \tilde{x}_1\left(\vec{b}\right), \tilde{x}_2\left(\vec{b}\right) \right\rangle$$
$$\vec{\tilde{x}}_1 \mathbin{\widetilde{+}} \vec{\tilde{x}}_2 := \lambda \vec{b}.\ \vec{\tilde{x}}_1\left(\vec{b}\right) + \vec{\tilde{x}}_2\left(\vec{b}\right)$$

$$\overset{\mathcal{V}}{\delta} : \forall R.\ \widehat{\mathcal{V}}^R \to \widetilde{\mathcal{V}}^R$$

$$\delta\left(\tilde{\iota}_\ell\right) := \tilde{\iota}_\ell$$
$$\delta\left(\tilde{\alpha}\right) := \tilde{\alpha}$$
$$\delta\left(\langle \widehat{v}_1, \widehat{v}_2 \rangle\right) := \delta\left(\widehat{v}_1\right) \otimes \delta\left(\widehat{v}_2\right)$$

$$\overset{env}{\delta} : \forall R.\ \widehat{env}^R \to \widetilde{env}^R \qquad \overset{env}{\delta}\left(\widehat{\gamma}\right) := \lambda \vec{b}.\ \left[x \mapsto \delta\left(\widehat{\gamma}(x)\right)\left(\vec{b}\right)\right]$$
$$\overset{store}{\delta} : \forall R.\ \widehat{store}^R \to \widetilde{store}^R \qquad \overset{store}{\delta}\left(\widehat{\sigma}\right) := \lambda \vec{b}.\ \left[\alpha \mapsto \left[n \mapsto \delta\left(\widehat{\sigma}(\alpha)[n]\right)\left(\vec{b}\right)\right]\right]$$

$$\mathcal{T}[\![\cdot]\!] : expr \to \forall R.\ \widehat{env}^R \times \widehat{store}^R \rightharpoonup \exists R'.\ \widetilde{trace}^{R'}$$

$$\mathcal{T}[\![a^l]\!]\left(\widehat{\gamma}, \widehat{\sigma}\right) := \lfloor l \rfloor \otimes \delta\left(\widehat{v}\right) \otimes \overset{env}{\delta}\left(\widehat{\gamma}'\right) \otimes \overset{store}{\delta}\left(\widehat{\sigma}'\right)$$
$$\textit{where}\quad \langle \widehat{v}, \widehat{\gamma}', \widehat{\sigma}' \rangle := \overset{atom}{\mathcal{E}}[\![a]\!]\left(\widehat{\gamma}, \widehat{\sigma}\right)$$
$$\mathcal{T}[\![\texttt{let}\ x := a^l\ \texttt{in}\ e]\!]\left(\widehat{\gamma}, \widehat{\sigma}\right) := \left(\lfloor l \rfloor \otimes \delta\left(\widehat{v}\right) \otimes \overset{env}{\delta}\left(\widehat{\gamma}'\right) \otimes \overset{store}{\delta}\left(\widehat{\sigma}'\right)\right) \mathbin{\widetilde{+}} t$$
$$\textit{where}\quad \langle \widehat{v}, \widehat{\gamma}', \widehat{\sigma}' \rangle := \overset{atom}{\mathcal{E}}[\![a]\!]\left(\widehat{\gamma}, \widehat{\sigma}\right) \quad t := \mathcal{T}[\![e]\!]\left(\widehat{\gamma}'\left[x \mapsto \widehat{v}\right], \widehat{\sigma}'\right)$$
$$\mathcal{T}[\![\texttt{let}\ x_1, x_2 := a^l\ \texttt{in}\ e]\!]\left(\widehat{\gamma}, \widehat{\sigma}\right) := \left(\lfloor l \rfloor \otimes \delta\left(\widehat{v}\right) \otimes \overset{env}{\delta}\left(\widehat{\gamma}'\right) \otimes \overset{store}{\delta}\left(\widehat{\sigma}'\right)\right) \mathbin{\widetilde{+}} t$$
$$\textit{where}\quad \langle \langle \widehat{v}_1, \widehat{v}_2 \rangle, \widehat{\gamma}', \widehat{\sigma}' \rangle := \overset{atom}{\mathcal{E}}[\![a]\!]\left(\widehat{\gamma}, \widehat{\sigma}\right) \quad t := \mathcal{T}[\![e]\!]\left(\widehat{\gamma}'\left[x_1 \mapsto \widehat{v}_1, x_2 \mapsto \widehat{v}_2\right], \widehat{\sigma}'\right)$$
$$\mathcal{T}[\![\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]\left(\widehat{\gamma}, \widehat{\sigma}\right) := cond\left(\widehat{v}, \mathcal{T}[\![e_1]\!]\left(\widehat{\gamma}', \widehat{\sigma}\right), \mathcal{T}[\![e_2]\!]\left(\widehat{\gamma}', \widehat{\sigma}\right)\right)$$
$$\textit{where}\quad \langle \widehat{v}, \widehat{\gamma}' \rangle := \overset{pico}{\mathcal{E}}[\![p]\!]\left(\widehat{\gamma}, \widehat{\sigma}\right)$$

Fig. 12. Denotational Trace Semantics

$$w \in \mathcal{W} ::= \theta \mid \langle w, w \rangle \qquad \omega \in penv := var \rightharpoonup \mathcal{W} \qquad \psi \in pstore := addr \rightharpoonup \mathcal{W}$$

$$all : \forall R\, X.\ \widetilde{X}^R \to \wp(X) \qquad all\,(\tilde{x}) := \left\{ x \;\middle|\; \exists \vec{b}.\ \tilde{x}\left(\vec{b}\right) = x \right\}$$

$$\begin{aligned}
&\overset{pico}{\mathcal{D}}[\![\cdot]\!] : \ pico \to \qquad penv \rightharpoonup \mathcal{W} \\
&\overset{atom}{\mathcal{D}}[\![\cdot]\!] : atom \to \forall R.\ \widehat{env}^R \times \widehat{store}^R \times penv \times pstore \times dep \rightharpoonup \mathcal{W} \times pstore \times dep \\
&\overset{exp}{\mathcal{D}}[\![\cdot]\!] : \ exp \to \forall R.\ \widehat{env}^R \times \widehat{store}^R \times penv \times pstore \times dep \rightharpoonup \mathcal{W} \times pstore \times dep
\end{aligned}$$

$$\overset{pico}{\mathcal{D}}[\![x]\!](\omega) := \omega(x) \qquad \overset{pico}{\mathcal{D}}[\![l]\!](\omega) := \varnothing$$

$$\begin{aligned}
\overset{atom}{\mathcal{D}}[\![p]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \left\langle \overset{pico}{\mathcal{D}}[\![p]\!](\omega), \psi, \theta \right\rangle \\
\overset{atom}{\mathcal{D}}[\![p^\circ]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \left\langle \overset{pico}{\mathcal{D}}[\![p]\!](\omega), \psi, \theta \right\rangle \\
\overset{atom}{\mathcal{D}}[\![p_1 \odot p_2]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \left\langle \overset{pico}{\mathcal{D}}[\![p_1]\!](\omega) \sqcup \overset{pico}{\mathcal{D}}[\![p_2]\!](\omega), \psi, \theta \right\rangle \\
\overset{atom}{\mathcal{D}}[\![\mathtt{asnat}(p)]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \left\langle \overset{pico}{\mathcal{D}}[\![p]\!](\omega), \psi, \theta \right\rangle \\
\overset{atom}{\mathcal{D}}[\![\mathtt{asbit}(p)]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \left\langle \overset{pico}{\mathcal{D}}[\![p]\!](\omega), \psi, \theta \right\rangle \\
\overset{atom}{\mathcal{D}}[\![\mathtt{flip}_\ell^\rho]\!]\left(\widehat{\gamma}^R, \widehat{\sigma}^R, \omega, \psi, \theta\right) &:= \langle \{R+1\}, \psi, \theta \rangle \\
\overset{atom}{\mathcal{D}}[\![\mathtt{use}(x)]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \langle \omega(x), \psi, \theta \rangle \\
\overset{atom}{\mathcal{D}}[\![\mathtt{reveal}(x)]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \langle \omega(x), \psi, \theta \sqcup \omega(x) \rangle \\
\overset{atom}{\mathcal{D}}[\![\overset{\cup}{\mathtt{mux}}(p_1, p_2, p_3)]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \langle \langle w, w \rangle, \psi, \theta \rangle \\
&\quad \textit{where} \quad w := \overset{pico}{\mathcal{D}}[\![p_1]\!](\omega) \sqcup \overset{pico}{\mathcal{D}}[\![p_2]\!](\omega) \sqcup \overset{pico}{\mathcal{D}}[\![p_3]\!](\omega) \\
\overset{atom}{\mathcal{D}}[\![\overset{\wedge}{\mathtt{mux}}(p_1, p_2, p_3)]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \langle \langle w, w \rangle, \psi, \theta \rangle \\
&\quad \textit{where} \quad w := \overset{pico}{\mathcal{D}}[\![p_2]\!](\omega) \sqcup \overset{pico}{\mathcal{D}}[\![p_3]\!](\omega) \\
\overset{atom}{\mathcal{D}}[\![\langle p_1, p_2 \rangle]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \theta) &:= \left\langle \left\langle \overset{pico}{\mathcal{D}}[\![p_1]\!](\omega), \overset{pico}{\mathcal{D}}[\![p_2]\!](\omega) \right\rangle, \psi, \theta \right\rangle
\end{aligned}$$

Fig. 13. Dependency Semantics (1)

$$\overset{atom}{\mathcal{D}}[\![\texttt{array}^N(p_1,\ldots,p_N)]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta) \coloneqq \left\langle \varnothing, \psi\left[\alpha \mapsto \overset{pico}{\mathcal{D}}[\![p_1]\!](\omega) \sqcup \cdots \sqcup \overset{pico}{\mathcal{D}}[\![p_N]\!](\omega)\right], \theta \right\rangle$$
$$\textit{where} \quad \alpha \coloneqq \mathit{fresh}(\widehat{\sigma})$$

$$\overset{atom}{\mathcal{D}}[\![p_1[p_2]]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta) \coloneqq \left\langle \bigsqcup_{\alpha} \psi(\alpha), \psi, \theta \right\rangle$$
$$\textit{where} \quad \langle \widehat{v}, \widehat{\gamma}' \rangle \coloneqq \overset{pico}{\mathcal{E}}[\![p_1]\!]\,(\widehat{\gamma}) \quad \alpha \in \mathit{all}(\widehat{v})$$

$$\overset{atom}{\mathcal{D}}[\![p_1[p_2] \leftarrow p_3]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta) \coloneqq \left\langle \bigsqcup_{\alpha} \psi(\alpha), \psi \sqcup \left[\alpha \mapsto \overset{pico}{\mathcal{D}}[\![p_3]\!](\omega)\right], \theta \right\rangle$$
$$\textit{where} \quad \langle \widehat{v}, \widehat{\gamma}' \rangle \coloneqq \overset{pico}{\mathcal{E}}[\![p_1]\!]\,(\widehat{\gamma}) \quad \alpha \in \mathit{all}(\widehat{v})$$

$$\overset{exp}{\mathcal{D}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta) \coloneqq \overset{atom}{\mathcal{D}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma},\psi,\omega,\theta)$$

$$\overset{exp}{\mathcal{D}}[\![\texttt{let } x \coloneqq a \texttt{ in } e]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta) \coloneqq \overset{exp}{\mathcal{D}}[\![e]\!]\,(\widehat{\gamma}'',\widehat{\sigma}',\omega'',\psi',\theta')$$
$$\textit{where} \quad \langle \widehat{v}, \widehat{\gamma}', \widehat{\sigma}' \rangle \coloneqq \overset{atom}{\mathcal{E}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma}) \qquad \widehat{\gamma}'' \coloneqq \widehat{\gamma}'[x \mapsto \widehat{v}]$$
$$\langle w, \psi', \theta' \rangle \coloneqq \overset{atom}{\mathcal{D}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta) \qquad \omega'' \coloneqq \omega'[x \mapsto w]$$

$$\overset{exp}{\mathcal{D}}[\![\texttt{let } x_1, x_2 \coloneqq a \texttt{ in } e]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta) \coloneqq \overset{exp}{\mathcal{D}}[\![e]\!]\,(\widehat{\gamma}'',\widehat{\sigma}',\omega'',\psi',\theta')$$
$$\textit{where} \quad \langle\langle \widehat{v}_1, \widehat{v}_2 \rangle, \widehat{\gamma}', \widehat{\sigma}' \rangle \coloneqq \overset{atom}{\mathcal{E}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma}) \qquad \widehat{\gamma}'' \coloneqq \widehat{\gamma}'[x_1 \mapsto \widehat{v}_1, x_2 \mapsto \widehat{v}_2]$$
$$\langle\langle w_1, w_2 \rangle, \omega', \psi', \theta' \rangle \coloneqq \overset{atom}{\mathcal{D}}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\theta) \qquad \omega'' \coloneqq \omega'[x_1 \mapsto w_1, x_2 \mapsto w_2]$$

$$\overset{exp}{\mathcal{D}}[\![\texttt{if } p \texttt{ then } e_1 \texttt{ else } e_2]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta) \coloneqq \left\langle \overset{pico}{\mathcal{D}}[\![p]\!](\omega) \sqcup w_1 \sqcup w_2, \psi_1 \sqcup \psi_2, \theta_1 \sqcup \theta_2 \right\rangle$$
$$\textit{where} \quad \langle w_1, \psi_1, \theta_1 \rangle \coloneqq \overset{exp}{\mathcal{D}}[\![e_1]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta)$$
$$\langle w_2, \psi_2, \theta_2 \rangle \coloneqq \overset{exp}{\mathcal{D}}[\![e_2]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\theta)$$

Fig. 14. Dependency Semantics (2)

$$\xi \in \mathit{renv} := \mathit{region} \to \mathit{dep}$$

$$\overset{\mathit{atom}}{\mathcal{R}}[\![\cdot]\!] : \mathit{atom} \to \forall R.\ \widehat{\mathit{env}}^R \times \widehat{\mathit{store}}^R \times \mathit{renv} \rightharpoonup \mathit{renv}$$
$$\overset{\mathit{exp}}{\mathcal{R}}[\![\cdot]\!] : \quad \mathit{exp} \to \forall R.\ \widehat{\mathit{env}}^R \times \widehat{\mathit{store}}^R \times \mathit{renv} \rightharpoonup \mathit{renv}$$

$$\overset{\mathit{atom}}{\mathcal{R}}[\![\texttt{flip}^{\rho}_{\ell}]\!]\left(\widehat{\gamma}^R, \widehat{\sigma}^R, \omega, \psi, \xi\right) := \xi \sqcup [\rho' \mapsto \{R+1\} \mid \rho \sqsubseteq \rho']$$
$$\overset{\mathit{atom}}{\mathcal{R}}[\![\ \_\ ]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \omega, \psi, \xi) := \xi$$

$$\overset{\mathit{exp}}{\mathcal{R}}[\![a]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \xi) := \overset{\mathit{atom}}{\mathcal{R}}[\![a]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \xi)$$
$$\overset{\mathit{exp}}{\mathcal{R}}[\![\texttt{let}\ x := a\ \texttt{in}\ e]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \xi) := \overset{\mathit{exp}}{\mathcal{R}}[\![e]\!]\left(\widehat{\gamma}'', \widehat{\sigma}', \overset{\mathit{atom}}{\mathcal{R}}[\![a]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \xi)\right)$$
$$\textit{where}\quad \langle \widehat{v}, \widehat{\gamma}', \widehat{\sigma}' \rangle := \overset{\mathit{atom}}{\mathcal{E}}[\![a]\!]\,(\widehat{\gamma}, \widehat{\sigma}) \quad \widehat{\gamma}'' := \widehat{\gamma}'\,[x \mapsto \widehat{v}]$$
$$\overset{\mathit{exp}}{\mathcal{R}}[\![\texttt{let}\ x_1, x_2 := a\ \texttt{in}\ e]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \xi) := \overset{\mathit{exp}}{\mathcal{R}}[\![e]\!]\left(\widehat{\gamma}'', \widehat{\sigma}', \overset{\mathit{atom}}{\mathcal{R}}[\![a]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \xi)\right)$$
$$\textit{where}\quad \langle \langle \widehat{v}_1, \widehat{v}_2 \rangle, \widehat{\gamma}', \widehat{\sigma}' \rangle := \overset{\mathit{atom}}{\mathcal{E}}[\![a]\!]\,(\widehat{\gamma}, \widehat{\sigma}) \quad \widehat{\gamma}'' := \widehat{\gamma}'\,[x_1 \mapsto \widehat{v}_1, x_2 \mapsto \widehat{v}_2]$$
$$\overset{\mathit{exp}}{\mathcal{R}}[\![\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \xi) := \overset{\mathit{exp}}{\mathcal{R}}[\![e_1]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \xi) \sqcup \overset{\mathit{exp}}{\mathcal{R}}[\![e_2]\!]\,(\widehat{\gamma}, \widehat{\sigma}, \xi)$$

Fig. 15. Region Semantics

$$\boxed{\Sigma \vdash \widehat{v}^R : \tau}$$

$$\textsc{Bit}\ \frac{}{\Sigma \vdash \tilde{b}^R : \texttt{bit}} \qquad \textsc{Nat}\ \frac{}{\Sigma \vdash \tilde{n}^R : \texttt{nat}} \qquad \textsc{Flip}\ \frac{}{\Sigma \vdash \tilde{b}^R : \texttt{flip}}$$

$$\textsc{Arr}\ \frac{\forall \alpha \in all\left(\tilde{\alpha}^R\right).\ \Sigma(\alpha) = \tau}{\Sigma \vdash \tilde{\alpha}^R : \texttt{array}(\tau)} \qquad \textsc{Tup}\ \frac{\Sigma \vdash \widehat{v}_1^R : \tau_1 \qquad \Sigma \vdash \widehat{v}_2^R : \tau_2}{\Sigma \vdash \left\langle \widehat{v}_1^R, \widehat{v}_2^R \right\rangle : \tau_1 \times \tau_2}$$

$$\boxed{\widehat{v}^R \ltimes_{/\!/\Phi} w}$$

$$\textsc{Dist}\ \frac{\tilde{\imath}^R \propto_{/\!/\Phi} \theta}{\tilde{\imath}_\ell^R \ltimes_{/\!/\Phi} \theta} \qquad \textsc{Arr}\ \frac{\tilde{\alpha}^R \propto_{/\!/\Phi} \theta}{\tilde{\alpha}^R \ltimes_{/\!/\Phi} \theta} \qquad \textsc{Tup}\ \frac{\widehat{v}_1^R \ltimes_{/\!/\Phi} w_1 \qquad \widehat{v}_2^R \ltimes_{/\!/\Phi} w_2}{\left\langle \widehat{v}_1^R, \widehat{v}_2^R \right\rangle \ltimes_{/\!/\Phi} \langle w_1, w_2 \rangle}$$

$$\boxed{\xi \vdash \tau \ltimes w}$$

$$\textsc{Base}\ \frac{\theta \subseteq \xi(\rho)}{\xi \vdash \beta@\langle \ell, \rho \rangle \ltimes \theta} \qquad \textsc{Arr}\ \frac{}{\xi \vdash array(\tau) \ltimes \theta} \qquad \textsc{Tup}\ \frac{\xi \vdash \tau_1 \ltimes w_1 \qquad \xi \vdash \tau_2 \ltimes w_2}{\xi \vdash \langle \tau_1, \tau_2 \rangle \ltimes \langle w_1, w_2 \rangle}$$

$$\boxed{\Sigma\ ;\ \xi \vdash v : \tau \ltimes_{/\!/\Phi} w}$$

$$\frac{\Sigma \vdash \widehat{v}^R : \tau \qquad \widehat{v}^R \ltimes_{/\!/\Phi} w \qquad \xi \vdash \tau \ltimes w}{\Sigma\ ;\ \xi \vdash \widehat{v}^R : \tau \ltimes_{/\!/\Phi} w}$$

Fig. 16. Value Well-formedness

$\boxed{\Gamma\ ;\ \Sigma \vdash \widehat{\gamma}}$

$$\Gamma\ ;\ \Sigma \vdash \widehat{\gamma} \coloneqq \forall x.\ \ \Sigma \vdash \widehat{\gamma}(x) : \Gamma(x)$$

$\boxed{\widehat{\gamma} \ltimes_{/\!/\Phi} \omega}$

$$\widehat{\gamma} \ltimes \omega \coloneqq \forall x.\ \ \widehat{\gamma}(x) \ltimes_{/\!/\Phi} \omega(x)$$

$\boxed{\xi \vdash \Gamma \ltimes \omega}$

$$\xi \vdash \Gamma \ltimes \omega \coloneqq \forall x.\ \ \xi \vdash \Gamma(x) \ltimes \omega(x)$$

$\boxed{\Gamma\ ;\ \Sigma\ ;\ \xi \vdash \widehat{\gamma} \ltimes_{/\!/\Phi} \omega}$

$$\frac{\Gamma\ ;\ \Sigma \vdash \widehat{\gamma} \qquad \widehat{\gamma} \ltimes_{/\!/\Phi} \omega \qquad \xi \vdash \Gamma \ltimes \omega}{\Gamma\ ;\ \Sigma\ ;\ \xi \vdash \widehat{\gamma} \ltimes_{/\!/\Phi} \omega}$$

$\boxed{\Sigma \vdash \widehat{\sigma}}$

$$\Sigma \vdash \widehat{\sigma} \coloneqq \forall \alpha.\ \ \Sigma \vdash \widehat{\sigma}(\alpha) : \Sigma(\alpha)$$

$\boxed{\widehat{\sigma} \ltimes_{/\!/\Phi} \psi}$

$$\widehat{\sigma} \ltimes \psi \coloneqq \forall \alpha.\ \ \widehat{\sigma}(\alpha) \ltimes_{/\!/\Phi} \psi(\alpha)$$

$\boxed{\xi \vdash \Sigma \ltimes \psi}$

$$\xi \vdash \Sigma \ltimes \psi \coloneqq \forall \alpha.\ \ \xi \vdash \Sigma(\alpha\rfloor \ltimes \psi(\alpha)$$

$\boxed{\Sigma\ ;\ \xi \vdash \widehat{\sigma} \ltimes_{/\!/\Phi} \psi}$

$$\frac{\Sigma \vdash \widehat{\sigma} \qquad \widehat{\sigma} \ltimes_{/\!/\Phi} \psi \qquad \xi \vdash \Sigma \ltimes \psi}{\Sigma\ ;\ \xi \vdash \widehat{\sigma} \ltimes_{/\!/\Phi} \psi}$$

Fig. 17. Env and Store Well-formedness

$$z \in \quad loc ::= \square \mid \langle z, \bullet \rangle \mid \langle \bullet, z \rangle$$
$$\phi \in \quad cloc ::= \mathtt{env}\,(x, z) \mid \mathtt{store}\,(\alpha, z)$$
$$\pi \in \quad rloc ::= \mathtt{ret}\,(z) \mid \phi$$
$$\beta \in btype ::= \mathtt{bit} \mid \mathtt{nat} \mid \mathtt{flip}$$

$$\widehat{cxt}^R := \widehat{env}^R \times \widehat{store}^R$$
$$\widehat{rxt}^R := \widehat{\mathcal{V}}^R \times \widehat{env}^R \times \widehat{store}^R$$
$$tcxt := tenv \times tstore$$
$$trxt := type \times tenv \times tstore$$

$$\cdot@\cdot : \forall R.\ \widehat{\mathcal{V}}^R \times loc \rightharpoonup \widetilde{lit}^R$$

$$\tilde{\imath}_\ell@\square := \tilde{\imath}$$

$$\langle \widehat{v}, \_ \rangle @\langle z, \bullet \rangle := \widehat{v}@z$$
$$\langle \_, \widehat{v} \rangle @\langle \bullet, z \rangle := \widehat{v}@z$$

$$\overset{cxt}{\cdot@\cdot} : \forall R.\ \widehat{cxt}^R \times cloc \rightharpoonup \widetilde{lit}^R$$

$$\langle \widehat{\gamma}, \widehat{\sigma} \rangle \overset{cxt}{@} \mathtt{env}\,(x, z) := \widehat{\gamma}(x)@z$$
$$\langle \widehat{\gamma}, \widehat{\sigma} \rangle \overset{cxt}{@} \mathtt{store}\,(\alpha, z) := \widehat{\sigma}(\alpha)@z$$

$$\overset{rxt}{\cdot@\cdot} : \forall R.\ \widehat{rxt}^R \times rloc \rightharpoonup \widetilde{lit}^R$$

$$\langle \widehat{v}, \widehat{\gamma}, \widehat{\sigma} \rangle \overset{rxt}{@} \mathtt{ret}\,(z) := \widehat{v}@z$$
$$\langle \widehat{v}, \widehat{\gamma}, \widehat{\sigma} \rangle \overset{rxt}{@} \mathtt{env}\,(x, z) := \widehat{\gamma}(x)@z$$
$$\langle \widehat{v}, \widehat{\gamma}, \widehat{\sigma} \rangle \overset{rxt}{@} \mathtt{store}\,(\alpha, z) := \widehat{\sigma}(\alpha)@z$$

$$\overset{type}{\cdot@\cdot} : type \times loc \rightharpoonup btype$$

$$\mathtt{bit}^\rho_\ell \overset{type}{@} \square := \mathtt{bit}$$
$$\mathtt{nat}^\rho_\ell \overset{type}{@} \square := \mathtt{nat}$$
$$\mathtt{flip}^\rho \overset{type}{@} \square := \mathtt{flip}$$

$$\langle \tau, \_ \rangle \overset{type}{@} \langle z, \bullet \rangle := \tau \overset{type}{@} z$$
$$\langle \_, \tau \rangle \overset{type}{@} \langle \bullet, z \rangle := \tau \overset{type}{@} z$$

$$\overset{tcxt}{\cdot@\cdot} : tcxt \times cloc \rightharpoonup btype$$

$$\langle \Gamma, \Sigma \rangle \overset{tcxt}{@} \mathtt{env}\,(x, z) := \Gamma(x) \overset{type}{@} z$$
$$\langle \Gamma, \Sigma \rangle \overset{tcxt}{@} \mathtt{store}\,(\alpha, z) := \Sigma(\alpha) \overset{type}{@} z$$

$$\overset{trxt}{\cdot@\cdot} : \forall R.\ trxt \times rloc \rightharpoonup btype$$

$$\langle \tau, \Gamma, \Sigma \rangle \overset{trxt}{@} \mathtt{ret}\,(z) := \tau \overset{type}{@} z$$
$$\langle \tau, \Gamma, \Sigma \rangle \overset{trxt}{@} \mathtt{env}\,(x, z) := \Gamma(x) \overset{type}{@} z$$
$$\langle \tau, \Gamma, \Sigma \rangle \overset{trxt}{@} \mathtt{store}\,(\alpha, z) := \Sigma(\alpha) \overset{type}{@} z$$

Fig. 18. Locations

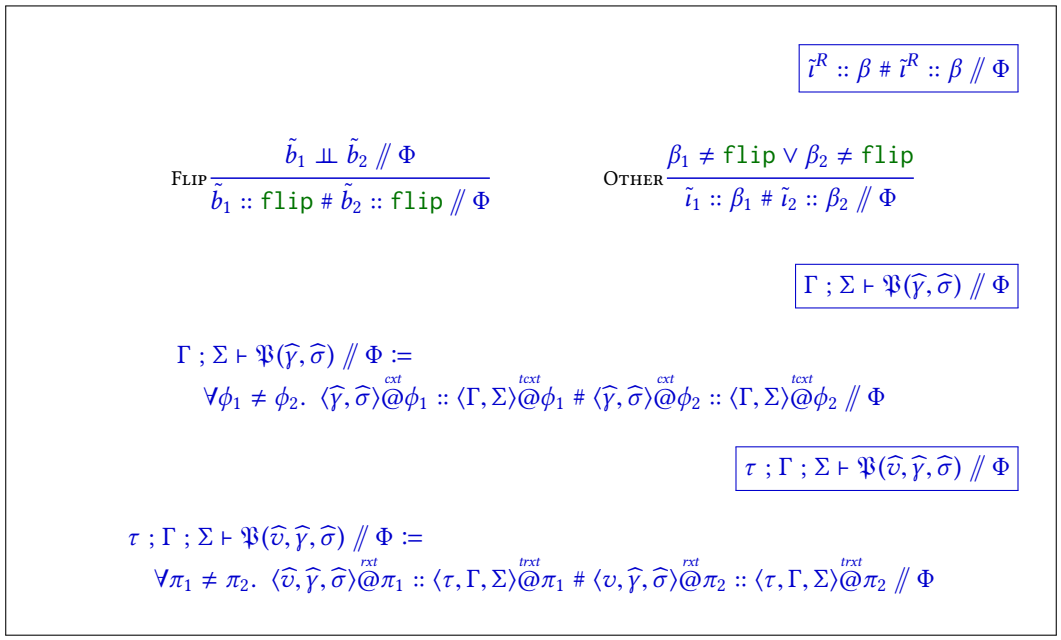

Fig. 19. Partitioning

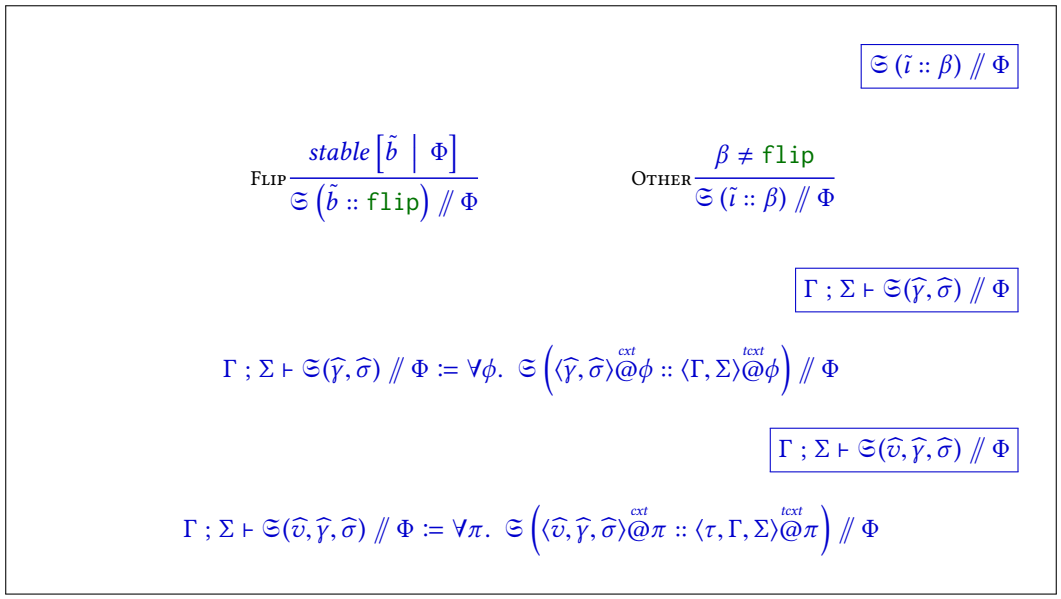

Fig. 20. Stability

$$
\begin{array}{rrcl}
l \in & loc & ::= & \ldots \textit{source code locations} \ldots \\
\dot{\iota} \in & \dot{lit} & ::= & b_{\mathsf{P}} \mid n_{\mathsf{P}} \mid \bullet \\
\dot{v} \in & \dot{\mathcal{V}} & ::= & \dot{\iota} \mid \alpha \mid \langle \dot{v}, \dot{v} \rangle \\
\dot{\gamma} \in & \dot{env} & := & var \rightharpoonup \dot{\mathcal{V}} \\
\dot{\sigma} \in & \dot{store} & := & addr \rightharpoonup \left(\dot{\mathcal{V}}\right)^* \\
\dot{\varsigma} \in & \dot{config} & := & \langle e, \dot{\gamma}, \dot{\sigma} \rangle \mid \mathsf{HALT}\,(\dot{v}, \dot{\gamma}, \dot{\sigma}) \\
\dot{t} \in & \dot{trace} & := & \dot{config}^*
\end{array}
\qquad
\begin{array}{rcl}
\widehat{\dot{v}}^R \in \widehat{\dot{\mathcal{V}}}^R & ::= & \widetilde{\dot{\iota}}^R \;\Big|\; \tilde{\alpha}^R \;\Big|\; \left\langle \widehat{\dot{v}}^R, \widehat{\dot{v}}^R \right\rangle \\
\widehat{\dot{\gamma}}^R \in \widehat{\dot{env}}^R & := & var \rightharpoonup \widehat{\dot{\mathcal{V}}} \\
\widehat{\dot{\sigma}}^R \in \widehat{\dot{store}}^R & := & addr \rightharpoonup \left(\widehat{\dot{\mathcal{V}}}\right)^*
\end{array}
$$

$$
\overset{lit}{\dot{obs}} : lit \times label \to \dot{lit} \qquad \overset{lit}{\dot{obs}}(\iota_{\mathsf{P}}) := \iota \qquad \overset{lit}{\dot{obs}}(\iota_{\mathsf{S}}) := \bullet
$$

$$
obs : \mathcal{V} \to \dot{\mathcal{V}} \qquad obs(\iota_\ell) := \overset{lit}{\dot{obs}}(\iota_\ell) \qquad obs(\alpha) := \alpha \qquad obs(\langle v_1, v_2 \rangle) := \langle obs(v_1), obs(v_2) \rangle
$$

$$
\begin{array}{rcl}
\overset{env}{\dot{obs}} : env \to \dot{env} & \qquad & \overset{env}{\dot{obs}}(\gamma)(v) := obs(\gamma(v)) \\
\overset{store}{\dot{obs}} : store \to \dot{store} & & \overset{store}{\dot{obs}}(\sigma)(\alpha)[n] := obs(\sigma(\alpha)[n])
\end{array}
$$

$$
\overset{config}{\dot{obs}} : config \to \dot{config}
$$

$$
\begin{array}{rcl}
\overset{config}{\dot{obs}}(\langle e, \gamma, \sigma \rangle) & := & \left\langle e, \overset{env}{\dot{obs}}(\gamma), \overset{env}{\dot{obs}}(\sigma) \right\rangle \\
\overset{config}{\dot{obs}}(\mathsf{HALT}\,(v, \gamma, \sigma)) & := & \mathsf{HALT}\left(\overset{\mathcal{V}}{\dot{obs}}(v), \overset{env}{\dot{obs}}(\gamma), \overset{env}{\dot{obs}}(\sigma)\right)
\end{array}
$$

$$
\overset{trace}{\dot{obs}} : config^* \to \dot{config}^* \qquad \overset{trace}{\dot{obs}}(\varsigma_1 \ldots \varsigma_N) := \overset{config}{\dot{obs}}(\varsigma_1) \ldots \overset{config}{\dot{obs}}(\varsigma_N)
$$

$$
\widehat{\dot{obs}} : \forall R.\ \widehat{\mathcal{V}}^R \to \widehat{\dot{\mathcal{V}}}^R \qquad \widehat{\dot{obs}}\,(\tilde{\iota}_\ell) := \lambda \vec{b}.\ \begin{cases} \iota & \text{if } \tilde{\iota}_\ell\left(\vec{b}\right) = \iota_{\mathsf{P}} \\ \bullet & \text{if } \tilde{\iota}_\ell\left(\vec{b}\right) = \iota_{\mathsf{S}} \end{cases}
$$

$$
\widehat{\dot{obs}}\,(\tilde{\alpha}) := \tilde{\alpha} \qquad \widehat{\dot{obs}}\left(\left\langle \widehat{\dot{v}}_1, \widehat{\dot{v}}_2 \right\rangle\right) := \left\langle \widehat{\dot{obs}}\left(\widehat{\dot{v}}_1\right), \widehat{\dot{obs}}\left(\widehat{\dot{v}}_2\right) \right\rangle
$$

$$
\begin{array}{rcl}
\overset{env}{\widehat{\dot{obs}}} : \forall R.\ \widehat{env}^R \to \widehat{\dot{env}}^R & \qquad & \overset{env}{\widehat{\dot{obs}}}\left(\widehat{\dot{\gamma}}\right)(v) := \widehat{\dot{obs}}\left(\widehat{\dot{\gamma}}(v)\right) \\
\overset{store}{\widehat{\dot{obs}}} : \forall R.\ \widehat{store}^R \to \widehat{\dot{store}}^R & & \overset{store}{\widehat{\dot{obs}}}\left(\widehat{\dot{\sigma}}\right)(\alpha)_n := \widehat{\dot{obs}}\left(\widehat{\dot{\sigma}}(\alpha)_n\right)
\end{array}
$$

Fig. 21. Adversary Observations

B THEOREMS

Theorem B.1 (Type Safety).

$$\textit{If} \quad \begin{array}{c} \Gamma\,;\,\Sigma \vdash a : \tau\,;\,\Gamma' \\ \Gamma\,;\,\Sigma \vdash \widehat{\gamma} \quad \Sigma \vdash \widehat{\sigma} \end{array} \quad \textit{then} \quad \exists \Sigma'\ \widehat{v}\ \widehat{\gamma}'\ \widehat{\sigma}'.\ \begin{array}{c} \mathcal{E}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma}) = \langle \widehat{v},\widehat{\gamma}',\widehat{\sigma}'\rangle \\ \Sigma' \vdash \widehat{v} : \tau \quad \Gamma'\,;\,\Sigma' \vdash \widehat{\gamma}' \quad \Sigma' \vdash \widehat{\sigma}' \end{array}$$

Definition B.2 (Well Typed Evaluation).

$$\boxed{\Gamma\,;\,\Sigma \vdash a : \tau\,;\,\Gamma'\,;\,\Sigma' \,;;\, \langle \widehat{\gamma},\widehat{\sigma}\rangle \Downarrow \langle \widehat{v},\widehat{\gamma}',\widehat{\sigma}'\rangle}$$

$$\frac{\begin{array}{c} \Gamma\,;\,\Sigma \vdash a : \tau\,;\,\Gamma' \qquad \Gamma\,;\,\Sigma \vdash \widehat{\gamma} \qquad \Sigma \vdash \widehat{\sigma} \\ \langle \widehat{v},\widehat{\gamma}',\widehat{\sigma}'\rangle := \mathcal{E}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma}) \qquad \Gamma'\,;\,\Sigma' \vdash \widehat{v} : \tau \qquad \Gamma'\,;\,\Sigma' \vdash \widehat{\gamma}' \qquad \Sigma' \vdash \widehat{\sigma}' \end{array}}{\Gamma\,;\,\Sigma \vdash a : \tau\,;\,\Gamma'\,;\,\Sigma' \,;;\, \langle \widehat{\gamma},\widehat{\sigma}\rangle \Downarrow \langle \widehat{v},\widehat{\gamma}',\widehat{\sigma}'\rangle}$$

Theorem B.3 (Dependency Soundness).

$$\textit{If} \quad \begin{array}{c} \Gamma\,;\,\Sigma \vdash a : \tau\,;\,\Gamma'\,;\,\Sigma' \,;;\, \langle \widehat{\gamma},\widehat{\sigma}\rangle \Downarrow \langle \widehat{v},\widehat{\gamma}',\widehat{\sigma}'\rangle \\ \widehat{\gamma} \ltimes \omega \qquad \widehat{\overset{env}{obs}}\,(\widehat{\gamma}) \ltimes \Phi \\ \widehat{\sigma} \ltimes \psi \qquad \widehat{\overset{store}{obs}}\,(\widehat{\sigma}) \ltimes \Phi \\ \mathcal{D}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\Phi) = \langle w,\psi',\Phi'\rangle \end{array} \quad \textit{then} \quad \begin{array}{ll} \widehat{v} \ltimes w & \widehat{obs}\,(\widehat{v}) \ltimes \Phi' \\ \widehat{\sigma}' \ltimes \psi' & \widehat{\overset{store}{obs}}\,(\widehat{\sigma}') \ltimes \Phi' \end{array}$$

Theorem B.4 (Region Soundness).

$$\textit{If} \quad \begin{array}{c} \Gamma\,;\,\Sigma \vdash a : \tau\,;\,\Gamma'\,;\,\Sigma' \,;;\, \langle \widehat{\gamma},\widehat{\sigma}\rangle \Downarrow \langle \widehat{v},\widehat{\gamma}',\widehat{\sigma}'\rangle \\ \xi \models \Gamma \ltimes \omega \qquad \widehat{\gamma} \ltimes_{/\!/\Phi} \omega \\ \xi \models \Sigma \ltimes \psi \qquad \widehat{\sigma} \ltimes_{/\!/\Phi} \psi \\ \mathcal{D}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\Phi) = \langle w,\psi',\Phi'\rangle \\ \mathcal{R}[\![a]\!]\,(\xi) = \xi' \end{array} \quad \textit{then} \quad \begin{array}{ll} \xi' \models \tau \ltimes w & \widehat{v} \ltimes_{/\!/\Phi'} w \\ \xi' \models \Sigma' \ltimes \psi' & \widehat{\sigma}' \ltimes_{/\!/\Phi'} \psi' \end{array}$$

Theorem B.5 (Well-partitioning).

$$\textit{If} \quad \begin{array}{c} \Gamma\,;\,\Sigma \vdash a : \tau\,;\,\Gamma'\,;\,\Sigma' \,;;\, \langle \widehat{\gamma},\widehat{\sigma}\rangle \Downarrow \langle \widehat{v},\widehat{\gamma}',\widehat{\sigma}'\rangle \\ \Gamma\,;\,\Sigma \vdash \mathfrak{P}\,(\widehat{\gamma},\widehat{\sigma}) \mathbin{/\!/} \Phi \\ \mathcal{D}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\Phi) = \langle w,\psi',\Phi'\rangle \end{array} \quad \textit{then} \quad \tau\,;\,\Gamma'\,;\,\Sigma' \vdash \mathfrak{P}\,(\widehat{v},\widehat{\gamma}',\widehat{\sigma}') \mathbin{/\!/} \Phi'$$

Theorem B.6 (Stability).

$$\textit{If} \quad \begin{array}{c} \Gamma\,;\,\Sigma \vdash a : \tau\,;\,\Gamma'\,;\,\Sigma' \,;;\, \langle \widehat{\gamma},\widehat{\sigma}\rangle \Downarrow \langle \widehat{v},\widehat{\gamma}',\widehat{\sigma}'\rangle \\ \Gamma\,;\,\Sigma \vdash \mathfrak{S}\,(\widehat{\gamma},\widehat{\sigma}) \mathbin{/\!/} \Phi \\ \mathcal{D}[\![a]\!]\,(\widehat{\gamma},\widehat{\sigma},\omega,\psi,\Phi) = \langle w,\psi',\Phi'\rangle \end{array} \quad \textit{then} \quad \tau\,;\,\Gamma'\,;\,\Sigma' \vdash \mathfrak{S}\,(\widehat{v},\widehat{\gamma}',\widehat{\sigma}') \mathbin{/\!/} \Phi'$$

Theorem B.7 (Strong Memory Trace Obliviousness (SMTO)).

$$\textit{If}\quad \begin{array}{c} \Gamma \vdash e : \tau \;;\; \Gamma' \\ \Gamma \;;\; \Sigma \vdash \widehat{\gamma}_i \qquad \widehat{\gamma}_i \ltimes_{/\!/\Phi} \omega \qquad \overset{env}{\widehat{obs}}\,(\widehat{\gamma}_i) \ltimes \Phi \qquad \xi \models \Gamma \ltimes \omega \\ \Sigma \vdash \widehat{\sigma}_i \qquad \widehat{\sigma}_i \ltimes_{/\!/\Phi} \psi \qquad \overset{store}{\widehat{obs}}\,(\widehat{\sigma}_i) \ltimes \Phi \qquad \xi \models \Sigma \ltimes \psi \\ \overset{env}{\widehat{obs}}\,(\widehat{\gamma}_1) \approx_{/\!/\Phi} \overset{env}{\widehat{obs}}\,(\widehat{\gamma}_2) \qquad \overset{store}{\widehat{obs}}\,(\widehat{\sigma}_1) \approx_{/\!/\Phi} \overset{store}{\widehat{obs}}\,(\widehat{\sigma}_2) \\ \mathfrak{P}\,(\widehat{\gamma}_i, \widehat{\sigma}_i) \qquad \mathfrak{S}\,(\widehat{\gamma}_i, \widehat{\sigma}_i) \end{array} \quad \textit{then} \quad \begin{array}{c} \overset{trace}{\widetilde{obs}}\,(\mathcal{T}\,[\![e]\!]\,(\widehat{\gamma}_1, \widehat{\sigma}_1)) \\ \approx_{/\!/\Phi} \\ \overset{trace}{\widetilde{obs}}\,(\mathcal{T}\,[\![e]\!]\,(\widehat{\gamma}_2, \widehat{\sigma}_2)) \end{array}$$

Corollary B.8 (Memory Trace Obliviousness (MTO)).

$$\textit{If}\quad \begin{array}{c} \Gamma \vdash e : \tau \;;\; \Gamma' \\ \Gamma \;;\; [] \vdash \lfloor \gamma_i \rfloor \\ \overset{env}{obs}\,(\gamma_1) = \overset{env}{obs}\,(\gamma_2) \end{array} \quad \textit{then} \quad \overset{trace}{\widetilde{obs}}\,(\mathcal{T}\,[\![e]\!]\,(\lfloor \gamma_1 \rfloor, [])) \approx \overset{trace}{\widetilde{obs}}\,(\mathcal{T}\,[\![e]\!]\,(\lfloor \gamma_2 \rfloor, []))$$

C LEMMAS REGARDING DISTRIBUTION MODEL

Lemma C.1 (Cond Probability Measure Semantics).

$$\mathfrak{p}[cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) = b] = \mathfrak{p}[\tilde{b}_1 = \mathtt{O} \wedge \tilde{b}_2 = b] + \mathfrak{p}[\tilde{b}_1 = \mathtt{I} \wedge \tilde{b}_3 = b]$$

Proof.

$$\begin{aligned}
&\mathfrak{p}[cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) = b] \\
&= \mathfrak{p}[cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) = b \wedge \tilde{b}_1 = \mathtt{O}] + \mathfrak{p}[cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) = b \wedge \tilde{b}_1 = \mathtt{I}] \\
&= \mathfrak{p}[cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) = b \mid \tilde{b}_1 = \mathtt{O}]\mathfrak{p}[\tilde{b}_1 = \mathtt{O}] + \mathfrak{p}[cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) = b \mid \tilde{b}_1 = \mathtt{I}]\mathfrak{p}[\tilde{b}_1 = \mathtt{I}] \\
&= \mathfrak{p}[\tilde{b}_2 = b \mid \tilde{b}_1 = \mathtt{O}]\mathfrak{p}[\tilde{b}_1 = \mathtt{O}] + \mathfrak{p}[\tilde{b}_3 = b \mid \tilde{b}_1 = \mathtt{I}]\mathfrak{p}[\tilde{b}_1 = \mathtt{I}] \\
&= \mathfrak{p}[\tilde{b}_2 = b \wedge \tilde{b}_1 = \mathtt{O}] + \mathfrak{p}[\tilde{b}_3 = b \wedge \tilde{b}_1 = \mathtt{I}] \\
&= \mathfrak{p}[\tilde{b}_1 = \mathtt{O} \wedge \tilde{b}_2 = b] + \mathfrak{p}[\tilde{b}_1 = \mathtt{I} \wedge \tilde{b}_3 = b]
\end{aligned}$$

□

Lemma C.2 (Cond Stability). *Assume* $\tilde{b}_1 \perp\!\!\!\perp \tilde{b}_2$, $\tilde{b}_1 \perp\!\!\!\perp \tilde{b}_3$ *and* $\tilde{b}_2 \approx \tilde{b}_3$. *Then:*

$$cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) \approx \tilde{b}_2 \approx \tilde{b}_3$$

Proof. Given b, the goal is to show $\mathfrak{p}[cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) = b] = \mathfrak{p}[\tilde{b}_2 = b] = \mathfrak{p}[\tilde{b}_3 = b]$.

$$\mathfrak{p}[cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) = b]$$

$$= \quad \wr \text{ Cond Probability Measure Semantics (Lemma (C.1))} \int$$

$$\mathfrak{p}[\tilde{b}_1 = \mathtt{O} \wedge \tilde{b}_2 = b] + \mathfrak{p}[\tilde{b}_1 = \mathtt{I} \wedge \tilde{b}_3 = b]$$

$$= \quad \wr \text{ Conditional independence } \tilde{b}_1 \perp\!\!\!\perp \tilde{b}_2 \text{ and } \tilde{b}_1 \perp\!\!\!\perp \tilde{b}_3 \int$$

$$\mathfrak{p}[\tilde{b}_1 = \mathtt{O}]\mathfrak{p}[\tilde{b}_2 = b] + \mathfrak{p}[\tilde{b}_1 = \mathtt{I}]\mathfrak{p}[\tilde{b}_3 = b]$$

$$= \quad \wr \text{ Distribution equivalence } \tilde{b}_2 \approx \tilde{b}_2 \int$$

$$\mathfrak{p}[\tilde{b}_1 = \mathtt{O}]\mathfrak{p}[\tilde{b}_2 = b] + \mathfrak{p}[\tilde{b}_1 = \mathtt{I}]\mathfrak{p}[\tilde{b}_2 = b]$$

$$= \quad \wr \text{ Common factor} \int$$

$$\mathfrak{p}[\tilde{b}_2 = b](\mathfrak{p}[\tilde{b}_1 = \mathtt{O}] + \mathfrak{p}[\tilde{b}_1 = \mathtt{I}])$$

$$= \quad \wr \text{ Proper Probability Measure (Lemma (C.5))} \int$$

$$\mathfrak{p}[\tilde{b}_2 = b] = \mathfrak{p}[\tilde{b}_3 = b]$$

□

Lemma C.3 (Probabilistic Independence). *Assume* $\tilde{b}_1 \ltimes \theta_1$, $\tilde{b}_2 \ltimes \theta_2$ *and* $\theta_1 \perp\!\!\!\perp \theta_2$. *Then:*

$$\tilde{b}_1 \perp\!\!\!\perp \tilde{b}_2$$

Lemma C.4 (Cond Independence). *Assume* $\tilde{b}_1 \ltimes \theta_1$, $\tilde{b}_2 \ltimes \theta_2$, $\tilde{b}_3 \ltimes \theta_3$, $\theta_1 \perp\!\!\!\perp \theta_2$, $\theta_1 \perp\!\!\!\perp \theta_3$ *and* $\tilde{b}_2 \approx \tilde{b}_3$. *Then:*

$$cond(\tilde{b}_1, \tilde{b}_2, \tilde{b}_3) \ltimes \theta_2 \cup \theta_3$$

Our model for probability measures allows for proving the following lemmas.

Lemma C.5 (Proper Probability Measure).

$$\mathfrak{p}[\tilde{b}_1 = \mathtt{O}] + \mathfrak{p}[\tilde{b}_2 = \mathtt{I}] = 1$$

Proof.

$$\begin{aligned}
&\mathfrak{p}[\tilde{b}_1 = \mathtt{O}] + \mathfrak{p}[\tilde{b}_2 = \mathtt{I}] \\
&= \frac{\mathfrak{c}[\tilde{b}_1 = \mathtt{O}]}{2^R} + \frac{\mathfrak{c}[\tilde{b}_1 = \mathtt{I}]}{2^R} \\
&= \frac{\left\|\{b' \mid \tilde{b}_1(b') = \mathtt{O}\}\right\| + \left\|\{b' \mid \tilde{b}_1(b') = \mathtt{I}\}\right\|}{2^R} \\
&= \frac{\left\|\{b' \mid \tilde{b}_1(b') = \mathtt{O} \vee \tilde{b}_1(b') = \mathtt{I}\}\right\|}{2^R} \\
&= \frac{\left\|\{b' \mid b' \in \mathbb{B}^R\}\right\|}{2^R} \\
&= \frac{2^R}{2^R} = 1
\end{aligned}$$

□

Lemma C.6 (Joint Probability Symmetry).

$$\mathfrak{p}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2] = \mathfrak{p}[\tilde{b}_2 = b_2 \wedge \tilde{b}_1 = b_1]$$

Proof is immediate.

Lemma C.7 (Chain Rule).

$$\mathfrak{p}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2] = \mathfrak{p}[\tilde{b}_1 = b_1 \mid \tilde{b}_2 = b_2]\mathfrak{p}[\tilde{b}_2 = b_2]$$

Proof.

$$\begin{aligned}
&\mathfrak{p}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2] \\
&= \frac{\mathfrak{c}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2]}{2^R} \\
&= \frac{\mathfrak{c}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2]}{\mathfrak{c}[\tilde{b}_2 = b_2]} \frac{\mathfrak{c}[\tilde{b}_2 = b_2]}{2^R} \\
&= \mathfrak{p}[\tilde{b}_1 = b_1 \mid \tilde{b}_2 = b_2]\mathfrak{p}[\tilde{b}_2 = b_2]
\end{aligned}$$

□

Corollary C.8 (Alt Chain Rule).

$$\mathfrak{p}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2] = \mathfrak{p}[\tilde{b}_2 = b_2 \mid \tilde{b}_1 = b_1]\mathfrak{p}[\tilde{b}_1 = b_1]$$

Consequence of Lemma (C.6).

Lemma C.9 (Bayes's Rule).

$$\mathfrak{p}[\tilde{b}_1 = b_1 \mid \tilde{b}_2 = b_2] = \frac{\mathfrak{p}[\tilde{b}_2 = b_2 \mid \tilde{b}_1 = b_1]\mathfrak{p}[\tilde{b}_1 = b_1]}{\mathfrak{p}[\tilde{b}_2 = b_2]}$$

Proof.

$$\begin{aligned}
&\mathfrak{p}[\tilde{b}_1 = b_1 \mid \tilde{b}_2 = b_2] \\
&= \frac{\mathfrak{c}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2]}{\mathfrak{c}[\tilde{b}_2 = b_2]} \\
&= \frac{\frac{\mathfrak{c}[\tilde{b}_2 = b_2 \wedge \tilde{b}_1 = b_1]}{\mathfrak{c}[\tilde{b}_1 = b_1]} \frac{\mathfrak{c}[\tilde{b}_1 = b_1]}{2^R}}{\frac{\mathfrak{c}[\tilde{b}_2 = b_2]}{2^R}} \\
&= \frac{\mathfrak{p}[\tilde{b}_2 = b_2 \mid \tilde{b}_1 = b_1]\mathfrak{p}[\tilde{b}_1 = b_1]}{\mathfrak{p}[\tilde{b}_2 = b_2]}
\end{aligned}$$

□

Corollary C.10 (Alt Bayes's Rule).

$$\mathfrak{p}[\tilde{b}_1 = b_1 \mid \tilde{b}_2 = b_2] = \frac{\mathfrak{p}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2]}{\mathfrak{p}[\tilde{b}_2 = b_2]}$$

Consequence of Lemma (C.7).

Lemma C.11 (Total Probability).

$$\mathfrak{p}[\tilde{b}_1 = b_1] = \sum_{b_2 \in \mathbb{B}} \mathfrak{p}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2]$$

Proof.

$$\mathfrak{p}[\tilde{b}_1 = b_1]$$

$$= \frac{\mathfrak{c}[\tilde{b}_1 = b_1]}{2^R}$$

$$= \frac{\left\|\{b' \mid \tilde{b}_1(b') = b_1\}\right\|}{2^R}$$

$$= \frac{\left\|\{b' \mid \tilde{b}_1(b') = b_1 \wedge (\tilde{b}_2 = \mathtt{O} \vee \tilde{b}_2 = \mathtt{I})\}\right\|}{2^R}$$

$$= \frac{\left\|\{b' \mid \tilde{b}_1(b') = b_1 \wedge \tilde{b}_2 = \mathtt{O} \vee \tilde{b}_1(b') = b_1 \wedge \tilde{b}_2 = \mathtt{I}\}\right\|}{2^R}$$

$$= \frac{\left\|\{b' \mid \tilde{b}_1(b') = b_1 \wedge \tilde{b}_2(b') = \mathtt{O}\}\right\| + \left\|\{b' \mid \tilde{b}_1(b') = b_1 \wedge \tilde{b}_2(b') = \mathtt{O}\}\right\|}{2^R}$$

$$= \frac{\mathfrak{c}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = \mathtt{O}]}{2^R} + \frac{\mathfrak{c}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = \mathtt{I}]}{2^R}$$

$$= \mathfrak{p}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = \mathtt{O}] + \mathfrak{p}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = \mathtt{I}]$$

$$\sum_{b_2 \in \mathbb{B}} \mathfrak{p}[\tilde{b}_1 = b_1 \wedge \tilde{b}_2 = b_2]$$

□