Oblivious Algebraic Data Types

QIANCHUAN YE, Purdue University, USA
BENJAMIN DELAWARE, Purdue University, USA

Secure computation allows multiple parties to compute joint functions over private data without leaking any sensitive data, typically using powerful cryptographic techniques. Writing secure applications using these techniques directly can be challenging, resulting in the development of several programming languages and compilers that aim to make secure computation accessible. Unfortunately, many of these languages either lack or have limited support for rich recursive data structures, like trees. In this paper, we propose a novel representation of structured data types, which we call oblivious algebraic data types, and a language for writing secure computations using them. This language combines dependent types with constructs for oblivious computation, and provides a security-type system which ensures that adversaries can learn nothing more than the result of a computation. Using this language, authors can write a single function over private data, and then easily build an equivalent secure computation according to a desired public view of their data.

CCS Concepts: • **Theory of computation** → **Type theory**; • **Software and its engineering** → **Data types and structures**; Functional languages; **Semantics**; • **Security and privacy** → *Logic and verification.*

Additional Key Words and Phrases: Dependent Types, Algebraic Data Types, Oblivious Computation, Multiparty Computation

ACM Reference Format:
Qianchuan Ye and Benjamin Delaware. 2022. Oblivious Algebraic Data Types. *Proc. ACM Program. Lang.* 6, POPL, Article 51 (January 2022), 29 pages. https://doi.org/10.1145/3498713

1 INTRODUCTION

It is often the case that the owners of some private data want to compute some joint function of their data: a group of hospitals, for example, may want to calculate some statistics about their patients. In the case that this data is sensitive, the parties may not want (or be legally allowed) to simply pool their data and compute the result. *Secure computation* provides a solution in such scenarios, allowing multiple parties to perform a joint computation while keeping their sensitive data secure. Secure computation was formally introduced by Yao [1982] in the early 1980s, and has since found many privacy-focused applications, including secure auctions, voting, and privacy-preserving machine learning [Evans et al. 2018; Hastings et al. 2019; Laud and Kamm 2015]. There are two major paradigms for secure computation: *secure multiparty computation* (MPC), wherein the computation is performed jointly by all parties involved; and *outsourced computation*, where a computationally powerful entity such as an untrusted cloud provider carries out the computation [Evans et al. 2018]. MPC is typically implemented using cryptography-based protocols, such as Yao's Garbled Circuits [Yao 1982] or secret-sharing [Goldreich et al. 1987], while outsourced computation can be implemented using a variety of mechanisms, including cryptography-based fully homomorphic encryption [Gentry 2009], virtualization [Barthe et al. 2014, 2019b] and secure processors [Hoekstra 2015].

Authors' addresses: Qianchuan Ye, Purdue University, USA, ye202@purdue.edu; Benjamin Delaware, Purdue University, USA, bendy@purdue.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2022 Copyright held by the owner/author(s).
2475-1421/2022/1-ART51
https://doi.org/10.1145/3498713

Writing secure applications which directly use these techniques can be quite challenging and error-prone, however, even if the author has the requisite cryptographic expertise. Thus, starting with Fairplay, the first publicly available MPC compiler [Malkhi et al. 2004], several high-level programming languages and compilers have been proposed to make secure computation accessible to non-experts [Hastings et al. 2019]. For example, Obliv-C [Zahur and Evans 2015] is a C-like language for MPC applications which compiles down to Yao's Garbled Circuits. Other notable languages include ObliVM [Liu et al. 2015], Wysteria/Wys* [Rastogi et al. 2014, 2019], λ_{obliv} [Darais et al. 2020].

Unfortunately, many of these languages either lack or have limited support for rich recursive data structures, like trees. When such data structures are supported, they typically require leaking information about the structure of the data: in Obliv-C, for example, users can define trees with secure nodes using pointers, but the "shape" of the underlying tree will always be visible to adversaries, as Obliv-C pointers are public data. In this paper, we propose a language that supports algebraic data structures capable of hiding not only their secure payload, but also their own structure. In this system, adversaries are not able to infer, say, whether an oblivious tree is left-heavy or right-heavy by observing the data structures themselves or how they are used.

One major obstacle to securely implementing such data structures is the possibility of *timing channels* in the programs that use them: the run time of any terminating computation reveals some approximate information about the "size" of the data structures it uses. Authors of secure computations must be careful to not inadvertently reveal more information through such timing side-channels. As an example, consider the following Obliv-C program, which traverses an oblivious array `a`:

```
for (i = 0; i < MAX_BOUND; i++) {
  // some secure computation on a[i]
}
```

Here, the author has chosen to avoid timing-channels by using a upper bound, `MAX_BOUND`, on the length of `a`. In effect, `MAX_BOUND` provides a sort of *public view* on the structure of `a`, which is then used to ensure a consistent running time for the `for` loop. Of course, the author also must ensure `a` has been padded out to this maximum bound and that there are no `break` statements that depend on the contents of `a` in the body of the loop. In order to be secure, a computation over structured data types must be carried out without revealing any information outside this public view, including the structure of the private data.

While this trick to make programs *constant-time* [Barthe et al. 2014] is easy to implement for computations over simple data types like arrays, it becomes more complicated for richer data structures. First, users have to decide how to manually "pad" data structures, so they are consistent with their public view. Second, programmers also have to track the public view throughout the program, making sure it remains consistent throughout. Lastly, richer data structures can have multiple public views, representing different trade-offs between privacy and performance. As an example, the public views of an oblivious tree include both its maximum depth and its spine. If the spine is known, less padding (and thus less "wasted" computation) is needed. Since the public view, like `MAX_BOUND` above, directly affects how programs are written, programmers are forced to write different versions of the same program even though the high-level program logic is exactly the same. Ideally, these concerns should be separated, allowing programmers to write the program once and for all, and then select the right public view for their security and performance requirements.

In this paper, we propose a novel representation of structured data types, which we call *oblivious algebraic data types* (OADTs). Our solution combines dependent types with language constructs for oblivious computation, and a security-type system which ensures that adversaries learn nothing

more than the output of the function and the input they provide. Using our language, clients can write a single function over private data, and then build an equivalent oblivious computation over some *public view* of that data. By switching views, users can explicitly trade off between how much information is leaked via public channels and the performance of the underlying computation.

In summary, this paper presents the following contributions:

- We observe that public views of private ADTs can be naturally expressed using dependent types with large elimination, allowing for a clean specification of what information is released at runtime.
- Exploiting this observation, we develop λ_{OADT}, a core calculus for writing oblivious programs using OADTs, whose strong type system ensures computations are secure.
- To enable both a more pleasant programming experience and more modular programs, we develop an extension of λ_{OADT}, dubbed $\lambda_{\mathsf{OADT}\boldsymbol{+}}$, which is equipped with a novel semantics which enable creating, from one single public program, oblivious programs with different public views.

Both λ_{OADT} and $\lambda_{\mathsf{OADT}\boldsymbol{+}}$ and their metatheory have been mechanically formalized in the Coq proof assistant. An artifact containing both these developments is publicly available [Ye and Delaware 2021].

2 OVERVIEW

```
data tree := leaf | node ℤ tree tree

def lookup (x : ℤ) (t : tree) : 𝔹 :=
  case t of
  leaf ⇒ false
  node y tl tr ⇒
      if x ≤ y
      then if y ≤ x
           then true
           else lookup x tl
      else lookup x tr
```

Fig. 1. Lookup element in a search tree

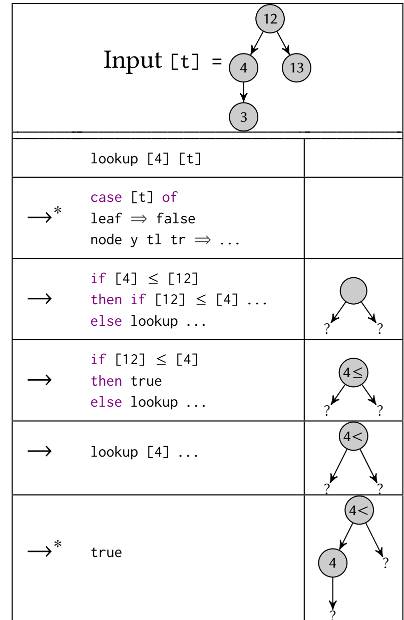


Fig. 2. Execution trace of lookup [4] [t]. The columns show the current state of the program, and information learned by the owner of the lookup key, respectively.

To illustrate our approach, consider the simple function in Figure 1, which looks for an element in a search tree by recursing over the tree. Suppose that Alice, the owner of a search tree, and Bob, the owner of some integer, want to check whether Bob's integer is a member of Alice's tree, without revealing any information to each other beyond what each can learn from their private data and the output. We adopt a variation of the standard semi-honest threat model from multiparty computation, where an untrusted party can observe every intermediate execution step of the program under a small-step operational semantics. Protecting against such a powerful attacker inevitably impacts the performance of secure applications, a point we will discuss in more detail at the end of this overview. For now, let us consider the implications of this attack model on our current example.

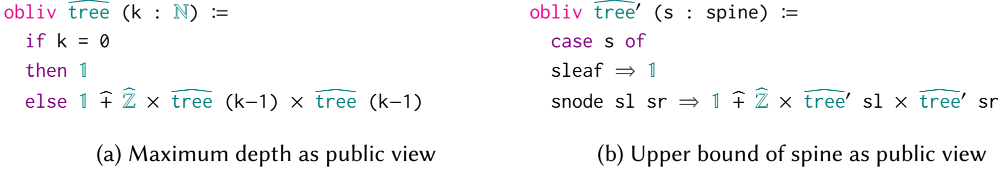


Fig. 3. Oblivious trees

Under this threat model, Bob can glean information about Alice's tree just by examining how it affects the control flow of the program, *even if the tree is perfectly obfuscated.* To see how, consider the execution trace of `lookup [4] [t]` shown in Figure 2, where `[t]` is the tree shown in the first row. The first column of each subsequent row shows the current execution step, while the second column shows what Bob can infer at that step. We use square brackets to denote that `[t]` is an *oblivious value*, i.e., it cannot be directly observed by a party. At each recursive call to `lookup`, there are two points that depend on the structure of the tree: the `case` statement that checks whether to recurse, and the `if` statement that decides which subtree to recurse on. As the fourth row illustrates, the branch `case` takes reveals some information about the structure of the current tree (it is non-empty) to Bob. The fifth and sixth row of the figure similarly show how the `if` statement reveals information about the relationship of the key to the value in the current node. By examining the program immediately following each such test, Bob adds to his knowledge of Alice's tree. At the end, Bob learns not only output of the function, but also a partial view of the tree's structure (including the exact node 4 is stored in); this view could be further refined by subsequent `lookup` operations.

Note that the participants of any terminating multiparty computation have to agree to share *some* public information: intuitively, simply knowing the number of intermediate steps in an execution of `lookup` leaks some upper bound on the number of nodes in the tree. Once that concession is made, the choice becomes *what* information to share– maybe the owner of the tree is okay with sharing its spine, but not the values stored in its internal nodes, or perhaps with revealing some upper bound on its depth[1]. The goal then, is to enable parties to compute functions over private data in a way that *only* depends on some mutually agreed upon public view of that data.

Oblivious Algebraic Data Types. The first component to our solution is our representation of both private data and its corresponding public view using oblivious algebraic data types. Figure 3a gives an example of an oblivious tree whose public view is its maximum depth. The private part of this type is built up from a set of oblivious type formers, e.g., $\widehat{\mathbb{Z}}$ and $\widehat{+}$ are the formers for oblivious fixed-width integers and oblivious coproducts, respectively. Section 3.4 formalizes oblivious data values, but the high-level intuition is that an observer of an execution trace cannot distinguish between the values of an oblivious type. When examining the trace in Figure 2, the oblivious integer `[4]` is *indistinguishable* from `[12]`, for example. The key idea behind oblivious ADTs is to construct a representation of private data from the public view. For example, all oblivious trees with a maximum depth of two are represented as:

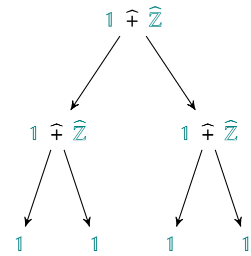


Fig. 4. Oblivious tree with a maximum depth of two

$$\widehat{\mathtt{tree}}\ 2 \equiv \mathbb{1} \mathbin{\widehat{+}} \widehat{\mathbb{Z}} \times (\mathbb{1} \mathbin{\widehat{+}} \widehat{\mathbb{Z}} \times \mathbb{1} \times \mathbb{1}) \times (\mathbb{1} \mathbin{\widehat{+}} \widehat{\mathbb{Z}} \times \mathbb{1} \times \mathbb{1})$$

[1] In the case the owner is okay with sharing the entire tree, the computation becomes quite efficient indeed!

Using data to compute a type is an example of *large elimination* from dependent type theory, where it is commonly used to recursively define propositions from terms. In this example, the type of an oblivious tree is computed from the public view `2`, resulting in the *type value* on the right hand side, which stipulates the "shape" of the private data. This type roughly corresponds to the tree shown in Figure 4. Every tree of this type is padded to depth 2, even a single "leaf", to avoid leaking structural information. This padding is implied by the use of oblivious coproduct $\hat{+}$, as the left injection and the right injection of an oblivious sum will be indistinguishable. The adversaries can not tell them apart by inspecting the payload, even if the two components have different types. The "tag" of a sum value is of course obfuscated as well. Constructing oblivious data types in this way ensures that all private values corresponding to a particular view are indistinguishable to an attacker.

Figure 3b shows the type of oblivious trees using an upper bound on its spine as the public view, where the spine is another user-defined ADT. This definition releases more public information than the one in Figure 3a, but it also enjoys a more efficient representation, as it requires less padding than a complete tree.

An Oblivious Language. Oblivious ADTs are only half the solution to secure computation– it still remains to ensure computations over private values are also oblivious. Even if an attacker cannot tell which values are being compared in $\texttt{if [4]} \le \texttt{[12]} \ldots$, they can still learn something about their relationship just by knowing the expression it steps to, as we saw in our previous example. To prevent these sorts of information leaks, we have designed λ_{OADT}, a pure functional language for writing secure computations over OADTs. λ_{OADT} is equipped with dependent types with large elimination to express OADTs, and type-based information flow control to guarantee oblivious computations.

Key to this calculus are its operations for securely constructing and destructing oblivious data values. As an example of these operations, consider the following λ_{OADT} expression, which compares two secure integers to determine what value to return:

$$\texttt{mux ([3]} \mathrel{\widehat{\leq}} \texttt{[4]) ([5]} \mathrel{\hat{+}_{\mathbb{Z}}} \texttt{[1]) ([6]} \mathrel{\hat{+}_{\mathbb{Z}}} \texttt{[1])}$$

We use $\hat{\cdot}$ to denote the secure versions of standard operations, such that $\texttt{[4]} \mathrel{\hat{+}_{\mathbb{Z}}} \texttt{[3]} \longrightarrow \texttt{[7]}$ and $\texttt{[4]} \mathrel{\widehat{\leq}} \texttt{[3]} \longrightarrow \texttt{[false]}$. Here, `mux` is a special conditional which returns an oblivious value according to the value of an oblivious boolean. In order to avoid leaking information, `mux` generates the same evaluation trace regardless of the value of the discriminee. To do so, it fully evaluates both branches before stepping to the final (oblivious) result:

$$\texttt{mux ([3]} \mathrel{\widehat{\leq}} \texttt{[4]) ([5]} \mathrel{\hat{+}_{\mathbb{Z}}} \texttt{[1]) ([6]} \mathrel{\hat{+}_{\mathbb{Z}}} \texttt{[1])} \longrightarrow^{*} \texttt{mux [true] [6] [7]} \longrightarrow \texttt{[6]}$$

Replacing `[3]` with `[5]` in the initial expression yields the same execution trace, modulo the private values at each step. Thus, nothing about the private information can be inferred by observing the execution:

$$\texttt{mux ([5]} \mathrel{\widehat{\leq}} \texttt{[4]) ([5]} \mathrel{\hat{+}_{\mathbb{Z}}} \texttt{[1]) ([6]} \mathrel{\hat{+}_{\mathbb{Z}}} \texttt{[1])} \longrightarrow^{*} \texttt{mux [false] [6] [7]} \longrightarrow \texttt{[7]}$$

The oblivious case statement behaves similarly, with the additional wrinkle that the pattern variables of the "wrong" branch are bound to some arbitrary oblivious values, which Section 3 explains in full detail.

λ_{OADT} is equipped with a security-type system [Sabelfeld and Myers 2003], to ensure the correct use of its secure operations. The full details of this type system can be found in Section 3.3, but at a high-level it enforces three key polices. First, oblivious types can only be built from oblivious types. For example, an oblivious coproduct cannot be built from public types, such as $\mathbb{B} \mathrel{\hat{+}} \mathbb{Z}$. If this were allowed, an adversary could infer whether a value of this type is a left or a right injection by observing the payload. Second, secure operations like `mux` can only be applied to oblivious terms. `mux [true] 1 2` is prohibited, for example, as knowing the public result of this `mux` reveals the

```
def lookup0 (x : ℤ̂) (k : ℕ) : treê k → 𝔹̂ :=
  if k = 0
  then λ _ ⇒ s𝔹 false
  else λ t ⇒ casê t of
             inl̂ _ ⇒ s𝔹 false
             inr̂ (y, tl, tr) ⇒
               mux (x ≦̂ y) (mux (y ≦̂ x) (s𝔹 true) (lookup0 x (k−1) tl)) (lookup0 x (k−1) tr)
```

Fig. 5. Oblivious lookup function in λ_{OADT}

oblivious discriminee. Third, types are treated as public information, otherwise the parties could not even agree on the data representation. Thus, oblivious types can only depend on public terms: `mux [true]` $\widehat{\mathbb{B}}$ $\widehat{\mathbb{Z}}$ is not a valid type in λ_{OADT}.

Figure 5 presents an oblivious implementation of the `lookup` function for the oblivious tree from Figure 3a. While the high-level program logic is the same, extra care is needed to ensure correct use of the oblivious tree. First, the function takes an extra argument for the public view; the argument needs to be correctly passed to every recursive call. Second, the function eliminates the public view, following the definition of $\widehat{\texttt{tree}}$, before accessing any secure data. Third, public constants and operations are replaced by their secure counterparts: e.g., `if` is replaced by `mux`. Similarly, the constants `true` and `false` are wrapped by the $\mathsf{s}_{\mathbb{B}}$ operation, which acts like a coercion from public booleans to oblivious booleans.

A More Ergonomic Oblivious Language. While the implementation of `lookup0` is guaranteed to be secure, it is quite far from the "standard" implementation of `lookup` in Figure 1, as its control flow has been restructured to only depend on public inputs and to meet the demands of a secure type system. As a consequence, a programmer must write distinct versions of `lookup` for each public view, despite the fact that the high-level program logic is exactly the same. Note that `lookup` is, in fact, a valid λ_{OADT} program, *as long as* it is applied to public data. This observation suggests the implementation of $\widehat{\texttt{lookup}}$ sketched to the right in Figure 6, which simply converts its private inputs to public versions, applies `lookup` to those arguments, and converts the result back to an oblivious value. From a cryptographic perspective, the arrow labeled $(\mathsf{r}_{\mathbb{Z}}, \mathsf{r}_{\texttt{tree}})$ corresponds to decrypting the secure inputs, while the arrow labeled $\mathsf{s}_{\mathbb{B}}$ corresponds to encryption. We refer to these two conversion functions as *retraction* and *section* to reflect their desired relationship: a retraction is a left inverse for section, i.e., if we apply the section function to a value, the retraction function converts the result back to the initial value. In other words, decrypting an encrypted value should return the original value. There is a fundamental flaw with this approach, however: applying a retraction in this manner completely leaks the private inputs of $\widehat{\texttt{lookup}}$! Thus, this program must be rejected by λ_{OADT}'s type system as insecure.

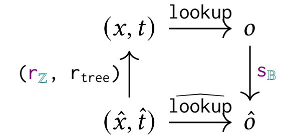
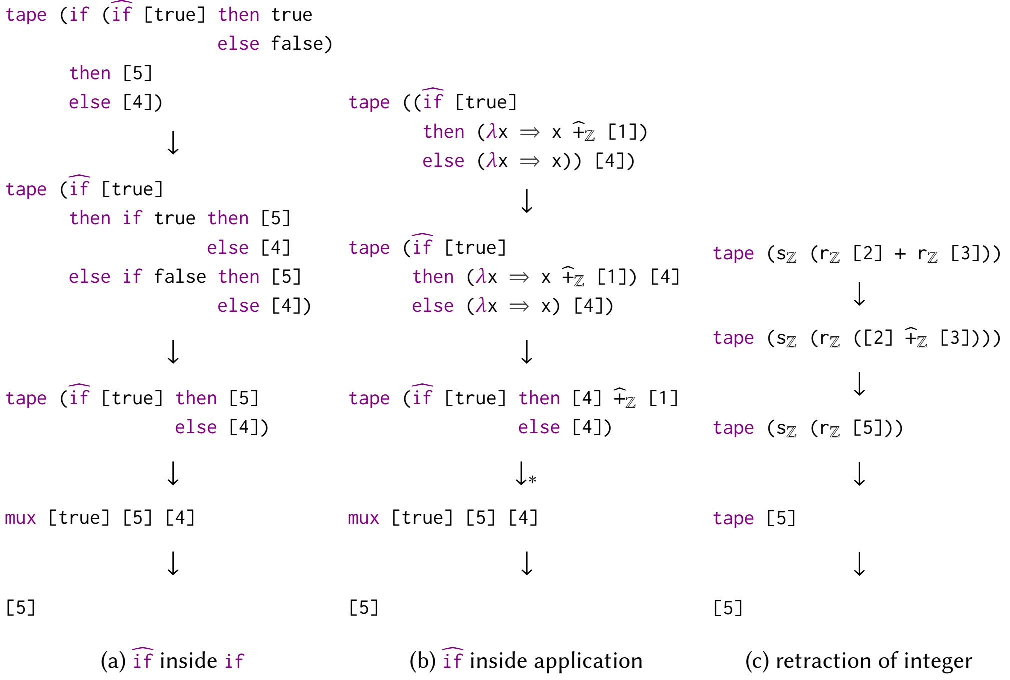


Fig. 6. Sketch of a secure `lookup` function

The ideal language for oblivious computation would permit implementations that combine the clarity of `lookup` with the security guarantees of `lookup0`. In pursuit of this goal, we have developed an extension of λ_{OADT}, called $\lambda_{\mathrm{OADT}\maltese}$, that allows implementations that follow the recipe sketched in Figure 6 *without* compromising obliviousness. Our key idea is to have the semantics of $\lambda_{\mathrm{OADT}\maltese}$ repair or "tape up" potentially leaky expressions during execution. This allows users to write section and leaky retraction functions that convert between oblivious and public values, relying on the semantics to ensure oblivious execution of any program that uses those functions.



(a) $\widehat{\text{if}}$ inside if (b) $\widehat{\text{if}}$ inside application (c) retraction of integer

Fig. 7. Example $\lambda_{\text{OADT}+}$ execution traces

To understand how this works, consider the execution trace of the simple $\lambda_{\text{OADT}+}$ program shown in Figure 7a. The new conditional $\widehat{\text{if}}$ is similar to mux in λ_{OADT}, but it allows non-oblivious branches. Note that this $\widehat{\text{if}}$ would leak the value of its private condition if it was evaluated using the semantics of mux. Similar leaks occur for any $\widehat{\text{if}}$ expression whose branches can evaluate to a public value. The idea behind the semantics of $\lambda_{\text{OADT}+}$ is straightforward: since $\widehat{\text{if}}$ only leaks information when it is evaluated, we will simply not do that! Rather, the surrounding tape annotation tells $\lambda_{\text{OADT}+}$ to defer reducing $\widehat{\text{if}}$ *until it is safe to do so*. This example makes progress by distributing the surrounding if statement into its branches and then evaluating both branches to oblivious values instead. Once both branches of an $\widehat{\text{if}}$ are evaluated to oblivious values, it can be securely reduced to a mux to produce the final result. Note that swapping [true] with [false] in this example produces the exact same trace, modulo oblivious values.

This example demonstrates the two key ideas behind the semantics of $\lambda_{\text{OADT}+}$: avoid leaks by delaying evaluation of potentially insecure expressions, while still making progress by distributing the surrounding context into such expressions. This strategy works for contexts like function application as well, as the example in Figure 7b shows. Figure 7c includes an example of a potential leak of oblivious integers via the $r_{\mathbb{Z}}$ operation, a leak that is ultimately patched using $s_{\mathbb{Z}}$. The program first progresses by distributing the insecure addition operation into retraction, then obliviously adding the result. After evaluating the oblivious addition, we have [5], and $s_{\mathbb{Z}}$ and $r_{\mathbb{Z}}$ can "cancel" each other, as the functions are effectively inverses. As [5] is already an oblivious value, tape becomes a no-op in this example.

Figure 8 shows the section and retraction functions for $\widehat{\text{tree}}$, along with a version of $\widehat{\text{lookup}}$ implemented using the recipe from Figure 6. While the section function is not used in $\widehat{\text{lookup}}$, it is needed for functions that return an oblivious tree. Function definitions in $\lambda_{\text{OADT}+}$ require an

```
def s_tree {⊥} (t : tree)⊤ (k : ℕ)⊥ : tree^ k ≔
  if k = 0
  then ()
  else tape (case t of
             leaf ⇒ inl^ ()
             node x tl tr ⇒ inr^ (tape (sℤ x, (s_tree tl (k−1)), (s_tree tr (k−1)))))

def r_tree {⊤} (k : ℕ)⊥ : (tree^ k)⊤ → tree ≔
  if k = 0
  then λ _ ⇒ leaf
  else λ t ⇒ case^ t of
             inl^ _ ⇒ leaf
             inr^ (x, tl, tr) ⇒ node (rℤ x) (r_tree (k−1) tl) (r_tree (k−1) tr)

def lookup^ {⊥} (x : ℤ^)⊤ (k : ℕ)⊥ (t : tree^ k)⊤ : 𝔹^ ≔
  s𝔹 (lookup (rℤ x) (r_tree k t))
```

Fig. 8. Oblivious lookup function in $\lambda_{\text{OADT}\maltese}$

additional annotation which signals if the function body includes any potentially leaky operations (e.g., $\widehat{\texttt{if}}$) that needs to be patched by the context surrounding the function call. Section 4 discusses how the type system of $\lambda_{\text{OADT}\maltese}$ uses these annotations in more detail. At a high level, its type system enforces two polices. First, as types are always public, they should not contain any potential leaks: any type which depends on $\widehat{\texttt{if}}$ [true] $\mathbb{1}$ $\widehat{\mathbb{B}}$ is disallowed, for example. Next, because only terms that evaluate to oblivious values can be patched up, our type system ensures that terms with potential leaks, e.g., a call to a retraction function or an $\widehat{\texttt{if}}$, are obliviously typed.

The Threat Model. Before presenting a detailed accounting of our calculi for oblivious computation, we pause to discuss the consequences of our chosen threat model. This strong threat model reflects both those of standard MPC protocols based on simultaneous execution of the program, e.g., secret-sharing [Beimel 2011; Goldreich et al. 1987], and those based on outsourced computation where the untrusted evaluators perform the execution, e.g., fully homomorphic encryption [Gentry 2009]. In these protocols, *any* party involved in the computation could be an attacker, including the ones executing the program, forcing us to protect against an attacker capable of observing the whole execution, including every intermediate program state. As a result, we have to obscure which branches the program takes. This threat model also naturally covers weaker adversaries, including those who can only observe the timing behavior.

Protecting against such a strong attacker necessarily comes with a cost: many of the asymptotic efficiency benefits normally enjoyed by ADTs are lost in the MPC setting. While the `lookup` function from our running example provides a simple and familiar illustration of OADTs, it is also not as performant as its insecure counterpart. In order to avoid leaking private information via control flow channels, `lookup` *must* touch all the elements in the tree— there is no way to implement a logarithmic oblivious lookup function for this particular OADT in $\lambda_{\text{OADT}\maltese}$. For fold-like computations that touch the entire data structure (e.g., map), however, the right choice of a public view (e.g., a tree whose spine is its public view) allows OADTs to feature similar asymptotic behavior to standard ADTs.

While sacrificing some performance gains for security, OADTs provide other advantages over unstructured data, much like their non-oblivious counterparts. OADTs enable users to more easily write computations over data that is naturally represented using ADTs, such as file systems,

```
e,τ  ::=                                            EXPRESSIONS:
      |  𝟙 | 𝔹 | 𝔹̂ | τ × τ | τ + τ | τ +̂ τ          simple types
      |  Πx:τ, τ                                    dependent function type
      |  x                                          variable
      |  () | true | false                          unit and boolean values
      |  λx:τ ⇒ e                                   function abstraction
      |  e e                                        expression and type application
      |  if e then e else e                         conditional
      |  mux e e e                                  atomic conditional
      |  (e, e) | π_b e                             pair and projection
      |  in_b<τ> e | in̂_b<τ> e                      (oblivious) sum injection
      |  case e of x ⇒ e | x ⇒ e                   sum elimination
      |  caŝe e of x ⇒ e | x ⇒ e                   oblivious sum elimination
      |  fold<X> e | unfold<X> e                    iso-recursive type intro. and elim.
      |  s_𝔹 e                                      section for boolean
      |  [b] | [in_b<ω̂> v̂]                          runtime boxed values
D    ::=                                            GLOBAL DEFINITIONS:
      |  data X := τ                                algebraic data type definition
      |  def x:τ := e                               (recursive) function definition
      |  obliv X̂ (x:τ) := τ                         (recursive) oblivious type definition
ω̂    ::=  𝟙 | 𝔹̂ | ω̂ × ω̂ | ω̂ +̂ ω̂                     OBLIVIOUS TYPE VALUES
v̂    ::=  () | [b] | (v̂, v̂) | [in_b<ω̂> v̂]             OBLIVIOUS VALUES
v    ::=  v̂ | b | (v, v) | λx:τ ⇒ e | in_b<τ> v | fold<X> v    VALUES
```

Fig. 9. λ_{OADT} syntax

organizational hierarchies, probability tree diagram, query languages, and decision trees. With OADTs, users can quickly prototype secure computations over structured data by simply writing their applications in a conventional functional language. Using the recipe illustrated by the $\widehat{\texttt{lookup}}$ function in Figure 8, programmers can use OADTs to explore the impact of different public views on a computation by simply providing different section and retraction functions.

3 λ_{OADT}, FORMALLY

This section formalizes λ_{OADT}, a core calculus for programming with OADTs. The calculi described in this section and the next have been mechanized in the Coq proof assistant; both developments can be found in the publicly available artifact [Ye and Delaware 2021].

3.1 Syntax

The core syntax of λ_{OADT} is shown in Figure 9. The full language, which can be found in the supplementary material, also includes let bindings. For simplicity, the core calculus of λ_{OADT} does not include primitive fixed-width integers. We discuss how the language may be extended with primitive integers in Section 4.5. As λ_{OADT} is dependently typed, types and terms belong to the same syntactic class, although by convention, we use the metavariable τ to refer to types, and e to terms. λ_{OADT} programs consist of an expression and a global context of public ADTs, oblivious ADTs, and functions. These are defined using `data`, `obliv`, and `def`, respectively. Using a global set of function definitions naturally supports general recursion and mutual recursion. When possible, we use lower case x for function names, upper case X for public ADT names, and $\widehat{X}$ for the names of oblivious ADTs.

Types in λ_{OADT} include dependent function types (Π), sums (+), products (×), and booleans ($\mathbb{B}$); as well as oblivious sums ($\widehat{+}$) and booleans ($\widehat{\mathbb{B}}$). We do not include a type for oblivious products, as they can be encoded via normal products with oblivious components. In λ_{OADT}, the typing rules and semantics for oblivious types are quite different from their public counterparts, which is why we choose to assign them distinct syntax, as opposed to using security labels [Zdancewic 2002]. λ_{OADT} supports type-level computation via large elimination, allowing users to compute types from terms using application, `if`, and `case`. Sum and product types are also allowed to have both oblivious and public components, allowing types to contain a mixture of public and private data.

Terms in λ_{OADT} are largely standard. A subscript distinguishes between left or right injection ($\mathtt{in}_b$) and projection (π_b), where the metavariable b is either `true` or `false`. We also use the more conventional synonyms `inl` (`inr`) and π_1 (π_2) for $\mathtt{in}_{\mathtt{true}}$ ($\mathtt{in}_{\mathtt{false}}$) and $\pi_{\mathtt{true}}$ ($\pi_{\mathtt{false}}$). Injections are annotated with their full type, in order to completely determine its data representation. λ_{OADT} has a nominal type system, so `fold` and `unfold` take the name of a public ADT, instead of a recursive type definition (i.e., μ type). `mux` is the core oblivious construct in λ_{OADT}; as Section 2 discussed, `mux` fully evaluates both of its branches before taking a single atomic step to the correct branch. Other oblivious constructs include boolean section $\mathtt{s}_{\mathbb{B}}$, which builds an oblivious boolean from its argument, and constructors ($\widehat{\mathtt{in}}_b$) and an eliminator ($\widehat{\mathtt{case}}$) for oblivious sums.

In addition to the expected sorts of public values, λ_{OADT} also includes oblivious values for booleans ($[\mathtt{b}]$) and sums ($[\mathtt{in}_b\langle\widehat{\omega}\rangle\ \widehat{\mathtt{v}}]$). In general, these oblivious values do not appear in the definitions in the global context: they are either provided by the data owner as the arguments to a global function at runtime, or created by evaluating $\mathtt{s}_{\mathbb{B}}$ or $\widehat{\mathtt{in}}_b$. As Section 2 discussed, these "boxed" values represent secure data which cannot be observed by an adversary. Since λ_{OADT} has type level computation, we also define a class of oblivious type values ($\widehat{\omega}$). Such values are built from a combination of oblivious base types and the other oblivious polynomial type formers.

3.2 Semantics

Figure 10 defines a relation for the small-step operational semantics of λ_{OADT}. The judgments of this relation have the form $\Sigma \vdash \mathtt{e} \longrightarrow \mathtt{e}'$, and are read as "e steps to e′ under the global context Σ". Since a λ_{OADT} program is evaluated under a fixed global context, we often abbreviate this judgment as $\mathtt{e} \longrightarrow \mathtt{e}'$, referring to Σ only when needed. The S-Ctx rule uses the evaluation contexts ($\mathcal{E}$) defined at the bottom of Figure 10 to evaluate subexpressions. While these evaluation contexts are not inductively defined, it is possible to recursively apply S-Ctx when evaluating subterms. This formulation is more convenient for the $\lambda_{\mathsf{OADT}✚}$ formalization in the next section.

Most of the non-oblivious reduction rules are standard, with some small deviations to keep the definitions succinct. In order to use the same syntax for both term-level and type-level application, for example, the rule for application evaluates the right expression before the left. In λ_{OADT}, the head of a type application is the global name of an oblivious type, which is not a value, so in order to avoid getting stuck when evaluating type-level application, we simply opt to evaluate arguments first. Adopting a dedicated syntax for type-level application would avoid this situation at the cost of some verbosity. In addition, several of the rules use the `ite` meta-function, which returns $\mathtt{e}_1$ when its first argument is `true`, and $\mathtt{e}_2$ otherwise. S-If is essentially the following two rules, for example:

S-If-True
$$\frac{}{\mathtt{if\ true\ then\ e_1\ else\ e_2} \longrightarrow \mathtt{e_1}}$$

S-If-False
$$\frac{}{\mathtt{if\ false\ then\ e_1\ else\ e_2} \longrightarrow \mathtt{e_2}}$$

To ensure that oblivious rules avoid leaking information, they require that any subexpressions have been fully evaluated before an oblivious expression is reduced. As an example, S-Ctx must be used to reduce the type and payload of an oblivious injection $\widehat{\mathtt{in}}$ to values before the S-OInj rule

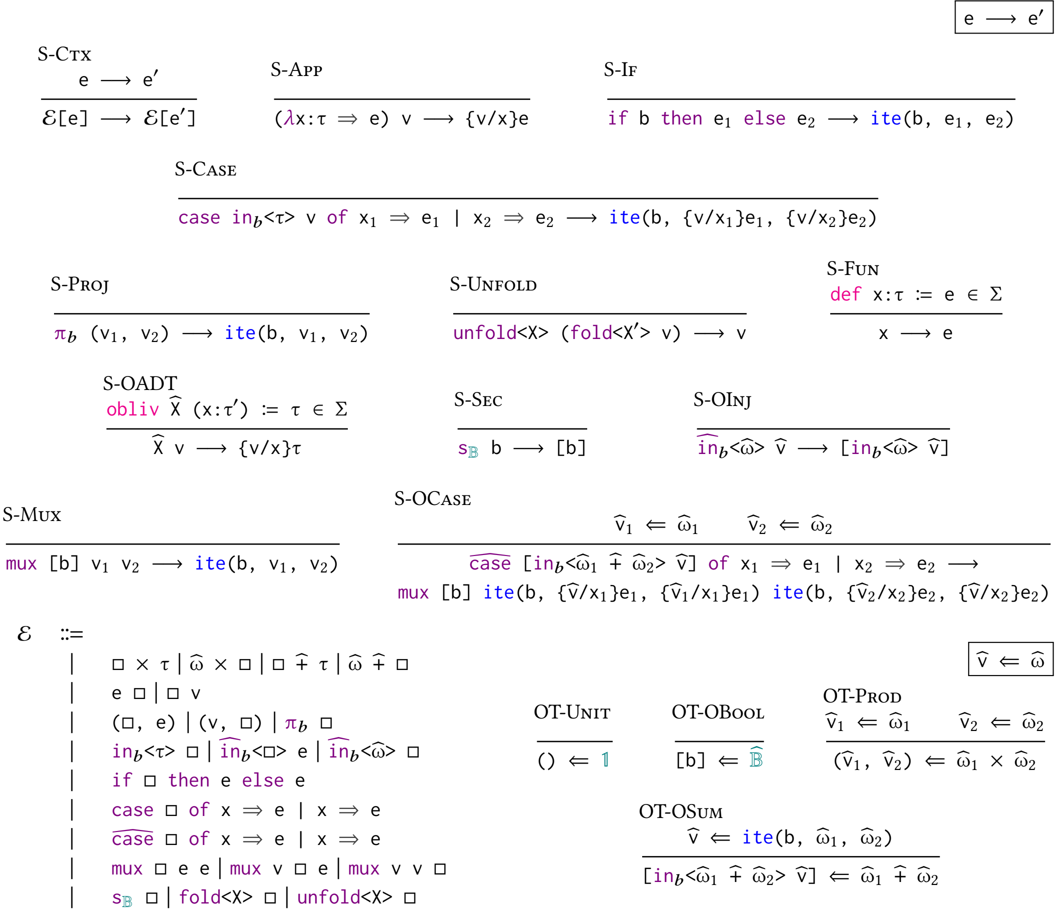


Fig. 10. λ_{OADT} semantics

can be applied to obtain the oblivious value. The other oblivious rules (e.g., S-Sec and S-Mux) are similar.

The most interesting evaluation rule is S-OCase, which also ensures that an adversary can not infer anything about the oblivious value being eliminated. In contrast to other oblivious elimination rules like S-Mux, each branch binds the value stored in the sum to its pattern variables. This begs the question of how to instantiate this variable when evaluating the "wrong" branch. Since this branch is eventually discarded when the resulting mux is evaluated, we opt to simply instantiate this variable with an arbitrary payload of the right type. This value is synthesized using the auxiliary relation, $\widehat{v} \Leftarrow \widehat{\omega}$, which is also shown in Figure 10. Equipped with this relation, S-OCase can be straightforwardly reduced to a mux expression. To see how, consider the rule corresponding to the case where b is false:

$$\frac{\widehat{v}_1 \Leftarrow \widehat{\omega}_1 \qquad \widehat{v}_2 \Leftarrow \widehat{\omega}_2}{\widehat{\mathtt{case}}\ [\mathtt{in}_{\mathtt{false}}\texttt{<}\widehat{\omega}_1 \mathbin{\widehat{+}} \widehat{\omega}_2\texttt{>}\ \widehat{v}]\ \mathtt{of}\ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \longrightarrow \mathtt{mux}\ [\mathtt{false}]\ \{\widehat{v}_1/x_1\}e_1\ \{\widehat{v}/x_2\}e_2}$$

In the true branch of the resulting mux expression, the pattern variable x_1 is instantiated with an arbitrary oblivious value, $\widehat{v}_1$, while the corresponding pattern variable in the false branch is

instantiated with the actual payload $\hat{v}$. Using this rule, the expression on the left below can step to either of the (indistinguishable) expressions on the right:

```
case [in_false<(B̂ × B̂) +̂ B̂> [false]] of        ⟶ mux [false] (π₂ ([false], [true])) [false]
  x₁ ⇒ π₂ x₁ | x₂ ⇒ x₂                          ⟶ mux [false] (π₂ ([true], [false])) [false]
```

3.3 Type System

The type system of λ_{OADT} ensures that well-typed programs are secure, in that adversaries cannot glean information about private data by observing public information. To guarantee this, kinds in λ_{OADT} are augmented with a *security label* which constrains how information flows through a program:

$$
\begin{array}{lcll}
\kappa & ::= & & \\
& | & *^{\mathsf{A}} & \text{Any} \\
& | & *^{\mathsf{P}} & \text{Public} \\
& | & *^{\mathsf{O}} & \text{Oblivious} \\
& | & *^{\mathsf{M}} & \text{Mixed}
\end{array}
$$

Types that can be treated as either public or oblivious are labeled with A. In practice, this is almost always the unit type, but it includes other singleton types, e.g., $\mathbb{1} \times \mathbb{1}$. Types which are entirely public or entirely private have the labels P and O, respectively. Finally, types with a mixture of public and private data, e.g., $\mathbb{B} \times \widehat{\mathbb{B}}$, are labeled with M. This label is also used to classify function types, which we will discuss in more detail shortly. Kinds form a secure join semi-lattice, as shown in Figure 11, with M being the most restrictive label. Unlike most secure type systems where types with a public label can be promoted to their secure counterparts, in λ_{OADT} public and oblivious labels are not compatible. We elide the security label of a kind when it is not relevant, e.g., $\Gamma \vdash \tau :: *$.

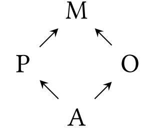


Fig. 11. Semi-lattice on λ_{OADT} kinds.

Programs in λ_{OADT} are typed using a pair of typing and kinding judgments; we denote these as $\Sigma;\ \Gamma \vdash e : \tau$ and $\Sigma;\ \Gamma \vdash \tau :: \kappa$, respectively. Figure 12 and Figure 13 give the kinding and typing rules for λ_{OADT}. We elide Σ from these definitions, as they both assume a fixed global context. For brevity, we omit some side conditions about kinding from the typing rules; these can be found in the Coq development.

The kinding rules for λ_{OADT} are shown in Figure 12. The rules for base types are straightforward. As previously mentioned, function types are assigned a mixed label. The reasons for this are two-fold: firstly, λ_{OADT} does not support oblivious function values. Secondly, this prevents function values from being used as the public view of oblivious types, making it easier for users to be sure oblivious types terminate. The subsumption rule K-Sub allows kinds to be converted to a more restricted label. This rule can be used with K-Prod to label a product type with the join of the labels of its components. K-Sum is similar, but it also includes the public label in the join, as the tag of a public sum is practically public. For example, $\mathbb{1} + \mathbb{1}$, which is equivalent to $\mathbb{B}$, should be kinded $*^{\mathsf{P}}$ instead of $*^{\mathsf{A}}$. Similarly, $\widehat{\mathbb{B}} + \widehat{\mathbb{B}}$ has to be kinded $*^{\mathsf{M}}$, the join of $*^{\mathsf{P}}$ and $*^{\mathsf{O}}$, as using it in an oblivious context risks leaking the tag. For similar reasons, K-OSum requires the components of oblivious sums to also be oblivious. K-OADT requires the argument of an oblivious type to be well-typed according to its definition in the global context. It does not need to check the index is public, as it is done when typing the global context. K-If and K-Case are the key components for large elimination. They both require the discriminee to be well-typed and the returned types to be obliviously kinded. The K-Case rule is rather permissive in that it does not require the type of discriminee e_0 to be completely publicly typed. While it is unclear when a programmer would ever

$$\boxed{\Gamma \vdash \tau :: \kappa}$$

$$\text{K-ADT}\ \frac{\texttt{data}\ X \coloneqq \tau \in \Sigma}{\Gamma \vdash X :: *^{P}} \qquad \text{K-UNIT}\ \frac{}{\Gamma \vdash \mathbb{1} :: *^{A}} \qquad \text{K-BOOL}\ \frac{}{\Gamma \vdash \mathbb{B} :: *^{P}} \qquad \text{K-OBOOL}\ \frac{}{\Gamma \vdash \widehat{\mathbb{B}} :: *^{O}}$$

$$\text{K-PI}\ \frac{\Gamma \vdash \tau_1 :: * \qquad x{:}\tau_1, \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \Pi x{:}\tau_1,\ \tau_2 :: *^{M}}$$

$$\text{K-PROD}\ \frac{\Gamma \vdash \tau_1 :: \kappa \qquad \Gamma \vdash \tau_2 :: \kappa}{\Gamma \vdash \tau_1 \times \tau_2 :: \kappa} \qquad \text{K-SUM}\ \frac{\Gamma \vdash \tau_1 :: \kappa \qquad \Gamma \vdash \tau_2 :: \kappa}{\Gamma \vdash \tau_1 + \tau_2 :: \kappa \sqcup *^{P}} \qquad \text{K-OSUM}\ \frac{\Gamma \vdash \tau_1 :: *^{O} \qquad \Gamma \vdash \tau_2 :: *^{O}}{\Gamma \vdash \tau_1 \mathbin{\widehat{+}} \tau_2 :: *^{O}}$$

$$\text{K-OADT}\ \frac{\texttt{obliv}\ \widehat{X}\ (x{:}\tau) \coloneqq \tau' \in \Sigma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \widehat{X}\ e :: *^{O}} \qquad \text{K-IF}\ \frac{\Gamma \vdash e_0 : \mathbb{B} \qquad \Gamma \vdash \tau_1 :: *^{O} \qquad \Gamma \vdash \tau_2 :: *^{O}}{\Gamma \vdash \texttt{if}\ e_0\ \texttt{then}\ \tau_1\ \texttt{else}\ \tau_2 :: *^{O}}$$

$$\text{K-CASE}\ \frac{\Gamma \vdash e_0 : \tau_1' + \tau_2' \qquad x_1{:}\tau_1', \Gamma \vdash \tau_1 :: *^{O} \qquad x_2{:}\tau_2', \Gamma \vdash \tau_2 :: *^{O}}{\Gamma \vdash \texttt{case}\ e_0\ \texttt{of}\ x_1 \Rightarrow \tau_1 \mid x_2 \Rightarrow \tau_2 :: *^{O}} \qquad \text{K-SUB}\ \frac{\Gamma \vdash \tau :: \kappa' \qquad \kappa' \sqsubseteq \kappa}{\Gamma \vdash \tau :: \kappa}$$

Fig. 12. λ_{OADT} kinding rules

actually use a type-level discriminee with oblivious components, it does not leak any information either.

The typing rules for public constructs are largely standard. Since λ_{OADT} is dependently typed, T-If and T-Case rely on an implicit motive that is specialized when typing branches[2]. This motive, (τ), has a special free variable (x) which stands in for the term being eliminated. The type used for the `then` branch in T-If ($\{\texttt{true}/x\}\tau$) concretizes the occurrences of this variable with `true`, for example. The typing rules for oblivious constructs are largely similar to their public counterparts, with the caveat that they place more constraints on their subterms: T-OInj requires the type of its payload to have an oblivious kind, for example. In addition to requiring that their branches have oblivious kind, the typing rules for oblivious eliminators (T-Mux and T-OCase) are required to return types that do not depend on the discriminees, in order to avoid leaking information about the discriminees via their types. T-BoxedLit and T-BoxedInj type oblivious values, with the latter simply outsourcing it to the relation used in S-OCase.

The final typing rule, T-Conv, allows any well-typed term to be typed using an equivalent type, denoted $\Sigma \vdash \tau \equiv \tau'$. This equivalence is defined directly in terms of a *parallel reduction* relation, $\Sigma \vdash e \Rrightarrow e'$, or simply $e \Rrightarrow e'$. Parallel reduction is a more liberal version of our call-by-value semantics which allows, for example, reduction under binders and congruence rules. Two terms are then said to be equivalent when they can parallel reduce to the same term in zero or more steps:

$$\Sigma \vdash \tau \equiv \tau' \triangleq \exists\, \tau''.\ \Sigma \vdash \tau \Rrightarrow^{*} \tau'' \land \Sigma \vdash \tau' \Rrightarrow^{*} \tau''$$

Parallel reduction also plays an important role in the metatheory of λ_{OADT}, particularly in the proof of obliviousness (Theorem 3.7).

A subset of the parallel reduction rules are shown in Figure 14; the remaining rules can be found in our Coq development; despite their importance in the metatheory of λ_{OADT}, the parallel reduction rules are straightforward. As the figure shows, the rules are essentially more permissive versions of their counterparts in the step relation from Figure 10. As an example, the parallel reduction rule

[2] This strategy is in line with other dependently typed languages (e.g., Coq), which try to infer a motive when none is supplied by the programmer.

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

$$\text{T-Var}\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{T-Unit}\ \frac{}{\Gamma \vdash () : \mathbb{1}} \qquad \text{T-Lit}\ \frac{}{\Gamma \vdash b : \mathbb{B}} \qquad \text{T-Fun}\ \frac{\mathtt{def}\ x{:}\tau \coloneqq e \in \Sigma}{\Gamma \vdash x : \tau} \qquad \text{T-Abs}\ \frac{x{:}\tau, \Gamma \vdash e : \tau' \qquad \Gamma \vdash \tau :: *}{\Gamma \vdash \lambda x{:}\tau \Rightarrow e : \Pi x{:}\tau,\ \tau'}$$

$$\text{T-App}\ \frac{\Gamma \vdash e_1 : \Pi x{:}\tau_2,\ \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \{e_2/x\}\tau_1} \qquad \text{T-Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1,\ e_2) : \tau_1 \times \tau_2} \qquad \text{T-Proj}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_b\ e : \mathtt{ite}(b,\ \tau_1,\ \tau_2)}$$

$$\text{T-Inj}\ \frac{\Gamma \vdash e : \mathtt{ite}(b,\ \tau_1,\ \tau_2) \qquad \Gamma \vdash \tau_1 + \tau_2 :: *}{\Gamma \vdash \mathtt{in}_b\texttt{<}\tau_1 + \tau_2\texttt{>}\ e : \tau_1 + \tau_2} \qquad \text{T-If}\ \frac{\Gamma \vdash e_0 : \mathbb{B} \qquad \Gamma \vdash e_1 : \{\mathtt{true}/x\}\tau \qquad \Gamma \vdash e_2 : \{\mathtt{false}/x\}\tau}{\Gamma \vdash \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \{e_0/x\}\tau}$$

$$\text{T-Case}\ \frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \qquad x_1{:}\tau_1, \Gamma \vdash e_1 : \{\mathtt{inl}\texttt{<}\tau_1{+}\tau_2\texttt{>}\ x_1/x\}\tau \qquad x_2{:}\tau_2, \Gamma \vdash e_2 : \{\mathtt{inr}\texttt{<}\tau_1{+}\tau_2\texttt{>}\ x_2/x\}\tau}{\Gamma \vdash \mathtt{case}\ e_0\ \mathtt{of}\ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : \{e_0/x\}\tau}$$

$$\text{T-Fold}\ \frac{\mathtt{data}\ X \coloneqq \tau \in \Sigma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{fold}\texttt{<}X\texttt{>}\ e : X} \qquad \text{T-Unfold}\ \frac{\mathtt{data}\ X \coloneqq \tau \in \Sigma \qquad \Gamma \vdash e : X}{\Gamma \vdash \mathtt{unfold}\texttt{<}X\texttt{>}\ e : \tau} \qquad \text{T-Sec}\ \frac{\Gamma \vdash e : \mathbb{B}}{\Gamma \vdash s_{\mathbb{B}}\ e : \widehat{\mathbb{B}}} \qquad \text{T-Mux}\ \frac{\Gamma \vdash e_0 : \widehat{\mathbb{B}} \qquad \Gamma \vdash \tau :: *^{O} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{mux}\ e_0\ e_1\ e_2 : \tau}$$

$$\text{T-OInj}\ \frac{\Gamma \vdash e : \mathtt{ite}(b,\ \tau_1,\ \tau_2) \qquad \Gamma \vdash \tau_1 \mathbin{\widehat{+}} \tau_2 :: *^{O}}{\Gamma \vdash \widehat{\mathtt{in}}_b\texttt{<}\tau_1 \mathbin{\widehat{+}} \tau_2\texttt{>}\ e : \tau_1 \mathbin{\widehat{+}} \tau_2} \qquad \text{T-OCase}\ \frac{\Gamma \vdash e_0 : \tau_1 \mathbin{\widehat{+}} \tau_2 \qquad \Gamma \vdash \tau :: *^{O} \qquad x_1{:}\tau_1, \Gamma \vdash e_1 : \tau \qquad x_2{:}\tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash \widehat{\mathtt{case}}\ e_0\ \mathtt{of}\ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : \tau}$$

$$\text{T-BoxedLit}\ \frac{}{\Gamma \vdash [b] : \widehat{\mathbb{B}}} \qquad \text{T-BoxedInj}\ \frac{[\mathtt{in}_b\texttt{<}\widehat{\omega}\texttt{>}\ \widehat{v}] \Leftarrow \widehat{\omega}}{\Gamma \vdash [\mathtt{in}_b\texttt{<}\widehat{\omega}\texttt{>}\ \widehat{v}] : \widehat{\omega}} \qquad \text{T-Conv}\ \frac{\Gamma \vdash e : \tau' \qquad \tau' \equiv \tau \qquad \Gamma \vdash \tau :: *}{\Gamma \vdash e : \tau}$$

Fig. 13. λ_{OADT} typing rules

for mux, R-Mux, does not reduce its branches to values, but immediately takes the corresponding branch, just like S-If. While this rule would leak information if the condition of the mux could be reduced to an oblivious value, this does not occur in practice. The reason for this is that parallel reduction is only used for statically type checking programs, and this rule will therefore never be used at runtime, when private data is made available.

To type a λ_{OADT} program, we also check the definitions in the global context using the rules in Figure 15. DT-Fun is straightforward: the type ascription needs to be well-kinded and the definition needs to be well-typed using an empty typing context. A definition may recursively refer to the name being defined, which is included in Σ. DT-ADT, the typing rule for public ADTs, simply requires the type to be completely public. The typing rule for oblivious ADTs, DT-OADT, requires that its index be completely public, as it is used as the public view. In contrast, the rest of the definition has to have an oblivious kind, under a context that includes the index x.

$$\boxed{e \Rrightarrow e'}$$

$$\text{R-Refl}\ \frac{}{e \Rrightarrow e} \qquad \text{R-App}\ \frac{e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2'}{(\lambda x{:}\tau \Rightarrow e_2)\ e_1 \Rrightarrow \{e_1'/x\}e_2'} \qquad \text{R-Fun}\ \frac{\mathtt{def}\ x{:}\tau \coloneqq e \in \Sigma}{x \Rrightarrow e} \qquad \text{R-OADT}\ \frac{\mathtt{obliv}\ \widehat{X}\ (x{:}\tau') \coloneqq \tau \in \Sigma \qquad e \Rrightarrow e'}{\widehat{X}\ e \Rrightarrow \{e'/x\}\tau}$$

$$\text{R-Mux}\ \frac{e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2'}{\mathtt{mux}\ [b]\ e_1\ e_2 \Rrightarrow \mathtt{ite}(b,\ e_1',\ e_2')} \qquad \text{R-Sec}\ \frac{}{s_{\mathbb{B}}\ b \Rrightarrow [b]} \qquad \text{R-OInj}\ \frac{}{\widehat{\mathtt{in}}_b\langle\widehat{\omega}\rangle\ \widehat{v} \Rrightarrow [\mathtt{in}_b\langle\widehat{\omega}\rangle\ \widehat{v}]}$$

$$\text{R-OCase}\ \frac{\widehat{v}_1 \Leftarrow \widehat{\omega}_1 \qquad \widehat{v}_2 \Leftarrow \widehat{\omega}_2 \qquad e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2'}{\begin{array}{c}\widehat{\mathtt{case}}\ [\mathtt{in}_b\langle\widehat{\omega}_1 \mathbin{\widehat{+}} \widehat{\omega}_2\rangle\ \widehat{v}]\ \mathtt{of}\ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 \Rrightarrow \\ \mathtt{mux}\ [b]\ \mathtt{ite}(b,\ \{\widehat{v}/x_1\}e_1',\ \{\widehat{v}_1/x_1\}e_1')\ \mathtt{ite}(b,\ \{\widehat{v}_2/x_2\}e_2',\ \{\widehat{v}/x_2\}e_2')\end{array}}$$

Fig. 14. Subset of $\lambda_{\mathtt{OADT}}$ parallel reduction rules

$$\boxed{\Sigma \vdash D}$$

$$\text{DT-Fun}\ \frac{\cdot \vdash \tau :: * \qquad \cdot \vdash e : \tau}{\Sigma \vdash \mathtt{def}\ x{:}\tau \coloneqq e} \qquad \text{DT-ADT}\ \frac{\cdot \vdash \tau :: *^{\mathsf{P}}}{\Sigma \vdash \mathtt{data}\ X \coloneqq \tau} \qquad \text{DT-OADT}\ \frac{\cdot \vdash \tau' :: *^{\mathsf{P}} \qquad x{:}\tau' \vdash \tau :: *^{\mathsf{O}}}{\Sigma \vdash \mathtt{obliv}\ \widehat{X}\ (x{:}\tau') \coloneqq \tau}$$

Fig. 15. $\lambda_{\mathtt{OADT}}$ global definition typing rules

3.4 Type Safety and Obliviousness

This section presents sketches of the key metatheory proofs for $\lambda_{\mathtt{OADT}}$'s type system. All the theorems in this section assume a well-typed global context. Firstly, $\lambda_{\mathtt{OADT}}$ enjoys the standard progress and preservation theorems:

Theorem 3.1 (Progress). *If* $\cdot \vdash e : \tau$*, then either* $e \longrightarrow e'$ *for some* e'*, or* e *is a value.*
If $\cdot \vdash \tau :: *^{\mathsf{O}}$*, then either* $\tau \longrightarrow \tau'$ *for some* τ'*, or* τ *is an oblivious type value.*

The proof of progress proceeds by mutual induction on typing and kinding derivation. The S-OCase case relies on the fact that every oblivious type value is inhabited, in order to find the oblivious value needed to reduce the "wrong" branch.

The preservation theorem also consists of two parts.

Theorem 3.2 (Preservation). *If* $\Gamma \vdash e : \tau$*, and* $e \longrightarrow e'$*, then* $\Gamma \vdash e' : \tau$*.*
If $\Gamma \vdash \tau :: \kappa$ *and* $\tau \longrightarrow \tau'$*, then* $\Gamma \vdash \tau' :: \kappa$*.*

The induction hypothesis for a direct proof of preservation is too weak to prove the T-If and T-Case cases. Instead, we show that the step relation refines parallel reduction and then prove preservation for the more general relation.

Lemma 3.3 (Preservation for parallel reduction). *If* $\Gamma \vdash e : \tau$*, and* $e \Rrightarrow e'$*, then* $\Gamma \vdash e' : \tau$*.*
If $\Gamma \vdash \tau :: \kappa$ *and* $\tau \Rrightarrow \tau'$*, then* $\Gamma \vdash \tau' :: \kappa$*.*

The proof of Lemma 3.3 depends on two additional lemmas. The first is a regularity lemma needed for the kinding constraints used by several typing rules.

Lemma 3.4 (Regularity). *If* $\Gamma \vdash \mathtt{e} : \tau$*, then* $\Gamma \vdash \tau :: \kappa$ *for some* κ*.*

The second is that parallel reduction is confluent.

Lemma 3.5 (Confluence of parallel reduction). *If* $\mathtt{e} \Rrightarrow^* \mathtt{e}_1$ *and* $\mathtt{e} \Rrightarrow^* \mathtt{e}_2$*, then there exists* $\mathtt{e}'$ *such that* $\mathtt{e}_1 \Rrightarrow^* \mathtt{e}'$ *and* $\mathtt{e}_2 \Rrightarrow^* \mathtt{e}'$*.*

Interestingly, the regular call-by-value semantics of λ_{OADT} are not confluent, thanks to a combination of the (limited) nondeterminism in S-OCase and nontermination. Observe that S-OCase can be applied with different choices of the arbitrary oblivious values. This is not a problem if the oblivious case expression terminates because the "wrong" branch will eventually be discarded. However, it is possible that the `mux` expression it steps to loops forever, such that the "wrong" branch is never discarded. Thankfully, the R-Mux rule is more liberal than S-Mux, ensuring that parallel reduction is confluent. Whenever $\widehat{\mathtt{case}}$ parallel reduces to a `mux` expression, however, R-Mux will immediately discard the "wrong" branch, forcing both choices to converge within one step.

3.4.1 Obliviousness. Adversaries should not be able to infer any information about the private information (i.e., oblivious values) of well-typed λ_{OADT} programs by observing the whole execution of a λ_{OADT} program. To prove this, we first formalize a notion of *indistinguishability* for λ_{OADT} expressions:

Definition 3.6 (Indistinguishability). We say two expressions are *indistinguishable*, denoted by $\mathtt{e} \approx \mathtt{e}'$, if

(1) they are both oblivious boolean values: $[\mathtt{b}] \approx [\mathtt{b}']$, or
(2) they are both oblivious injections with the same type: $[\mathtt{in}_b\texttt{<}\widehat{\omega}\texttt{>}\ \mathtt{v}] \approx [\mathtt{in}_{b'}\texttt{<}\widehat{\omega}\texttt{>}\ \mathtt{v}']$, or
(3) they are the same expression with indistinguishable sub-expressions.

Intuitively, two expressions are indistinguishable if they only differ in their oblivious values. Note that indistinguishability is a completely syntactic notion: two lambda abstractions are indistinguishable only if their bodies are indistinguishable. This is a direct consequence of our strong threat model: dishonest parties are capable of peeking "under the binders", i.e., lambda abstractions are *not* black boxes to them. As an example, the functions $\lambda \mathtt{x}\ \mathtt{y} \Rightarrow \mathtt{x} + \mathtt{y}$ and $\lambda \mathtt{x}\ \mathtt{y} \Rightarrow \mathtt{y} + \mathtt{x}$ are not indistinguishable, even though their "big-step" behaviors are the same: if `mux [true] (λx y ⇒ x + y) (λx y ⇒ y + x)` were to step to $\lambda \mathtt{x}\ \mathtt{y} \Rightarrow \mathtt{x} + \mathtt{y}$, an attacker could learn about the private condition by inspecting the resulting function. More pleasantly, this syntactic definition enjoys a congruence property: plugging indistinguishable partial programs into indistinguishable contexts is guaranteed to result in indistinguishable whole programs.

Equipped with this relation, we can now formally state the obliviousness theorem for λ_{OADT}:

Theorem 3.7 (Obliviousness). *If* $\mathtt{e}_1 \approx \mathtt{e}_2$ *and* $\cdot \vdash \mathtt{e}_1 : \tau_1$ *and* $\cdot \vdash \mathtt{e}_2 : \tau_2$*, then*

(1) $\mathtt{e}_1 \longrightarrow^n \mathtt{e}_1'$ *if and only if* $\mathtt{e}_2 \longrightarrow^n \mathtt{e}_2'$ *for some* $\mathtt{e}_2'$*.*
(2) *if* $\mathtt{e}_1 \longrightarrow^n \mathtt{e}_1'$ *and* $\mathtt{e}_2 \longrightarrow^n \mathtt{e}_2'$*, then* $\mathtt{e}_1' \approx \mathtt{e}_2'$*.*

We write $\mathtt{e} \longrightarrow^n \mathtt{e}'$ to mean $\mathtt{e}$ reduces to $\mathtt{e}'$ in exactly n steps. The first piece of this theorem is a generalization of progress, and ensures that information is not leaked via a termination channel. The second piece says that for any two indistinguishable programs, an observer cannot learn anything about their oblivious values by examining the states they can step to. Taken together, these two properties ensure that an observer cannot learn anything about the private values in a well-typed λ_{OADT} program, even given the entire execution trace of that program. If we treat the observable parts of the intermediate execution states as a public channel, obliviousness provides a sort of noninterference property [Goguen and Meseguer 1982; Sabelfeld and Myers 2003], in that different private (i.e., high-security) inputs do not leak any information via this public channel.

$\mathtt{e},\tau$	::=		EXTENDED EXPRESSIONS:
	|	...	
	|	$\widehat{\mathtt{if}}$ e then e else e	oblivious leaky conditional
	|	tape e	tape operation
	|	$\lambda \mathtt{x}:_l \tau \Rightarrow \mathtt{e} \mid \Pi \mathtt{x}:_l \tau,\ \tau$	function and function types with leakage label
D	::=		EXTENDED GLOBAL DEFINITIONS:
	|	...	
	|	def $\mathtt{x}:_l \tau$:= e	(recursive) function definition with leakage label
l	::=	$\top \mid \bot$	LEAKAGE LABEL

Fig. 16. $\lambda_{\mathsf{OADT}✚}$ syntax

The proof of Theorem 3.7 is by induction on the derivation of $\mathtt{e}_1 \longrightarrow^n \mathtt{e}_1'$. The first part of the proof of obliviousness is a direct consequence of progress and the fact that well-typed values are only indistinguishable from other values. The second part is more involved, and requires the following two key lemmas to prove the S-MUX case:

LEMMA 3.8. *If* $\Gamma \vdash \mathtt{v} : \tau$ *and* $\Gamma \vdash \mathtt{v}' : \tau$, *and* $\Gamma \vdash \tau :: *^{\mathsf{O}}$, *then* $\mathtt{v} \approx \mathtt{v}'$.

LEMMA 3.9. *If* $\mathtt{v} \approx \mathtt{v}'$, $\Gamma \vdash \mathtt{v} : \tau$, $\Gamma \vdash \mathtt{v}' : \tau'$, *and* $\Gamma \vdash \tau :: *^{\mathsf{O}}$, *then* $\tau \equiv \tau'$.

Lemma 3.8 states that all values of the same oblivious type are indistinguishable, and Lemma 3.9 ensures that two indistinguishable, obliviously-typed values have the same type up to type equivalence. The proofs of both lemmas proceed by induction on the typing derivation. Most of the proofs are straightforward, except for the case of T-CONV in both lemmas. Since applying the induction hypothesis requires that τ' also be oblivious, we need to show that two equivalent, well-kinded types simultaneously have oblivious kinds, which follows from Lemma 3.3:

LEMMA 3.10. *If* $\tau \equiv \tau'$, $\Gamma \vdash \tau :: *^{\mathsf{O}}$, *and* $\Gamma \vdash \tau' :: *$, *then* $\Gamma \vdash \tau' :: *^{\mathsf{O}}$.

In practice, well-typed λ_{OADT} programs are functions that take arguments of oblivious types, such as $\widehat{\mathsf{lookup}}$ from Figure 8. The program built by supplying such a function with private inputs of the right types is indistinguishable from one built using different private inputs, thanks to the congruence property of indistinguishability and Lemma 3.8. As a direct consequence of the obliviousness theorem, an attacker can not glean any information about the private inputs of such programs. This fact is captured in the following corollary about open λ_{OADT} programs:

COROLLARY 3.11. *If* $\mathtt{x}:\tau' \vdash \mathtt{e} : \tau$ *with* $\cdot \vdash \tau' :: *^{\mathsf{O}}$, *then for any two values* $\mathtt{v}_1$ *and* $\mathtt{v}_2$ *of oblivious type* τ'*:*

(1) $\{\mathtt{v}_1/\mathtt{x}\}\mathtt{e} \longrightarrow^n \mathtt{e}_1$ *if and only if* $\{\mathtt{v}_2/\mathtt{x}\}\mathtt{e} \longrightarrow^n \mathtt{e}_2$ *for some* $\mathtt{e}_2$.
(2) $\{\mathtt{v}_1/\mathtt{x}\}\mathtt{e} \longrightarrow^n \mathtt{e}_1$ *and* $\{\mathtt{v}_2/\mathtt{x}\}\mathtt{e} \longrightarrow^n \mathtt{e}_2$ *implies that* $\mathtt{e}_1$ *and* $\mathtt{e}_2$ *are indistinguishable, i.e.,* $\mathtt{e}_1 \approx \mathtt{e}_2$.

4 $\lambda_{\mathsf{OADT}✚}$, FORMALLY

This section formalizes $\lambda_{\mathsf{OADT}✚}$, an extension to λ_{OADT} that permits implementations in the vein of Figure 6.

4.1 Syntax

The extended syntax of $\lambda_{\mathsf{OADT}✚}$ is shown in Figure 16. These extensions permit $\lambda_{\mathsf{OADT}✚}$ expressions that *potentially* leak information *locally*, as long as they can eventually be repaired by the surrounding context. The new $\widehat{\mathtt{if}}$ operation is similar to mux, but its branches are permitted to be non-oblivious, causing a potential leak if $\widehat{\mathtt{if}}$ is evaluated naively. The new tape annotation acts as a boundary for

$$\boxed{\texttt{e} \longrightarrow \texttt{e}'}$$

S-OCase

$$\frac{\widehat{\texttt{v}}_1 \Leftarrow \widehat{\omega}_1 \qquad \widehat{\texttt{v}}_2 \Leftarrow \widehat{\omega}_2}{\begin{array}{c}\widehat{\texttt{case}}\ [\texttt{in}_b\texttt{<}\widehat{\omega}_1 \mathbin{\widehat{+}} \widehat{\omega}_2\texttt{>}\ \widehat{\texttt{v}}]\ \texttt{of}\ \texttt{x}_1 \Rightarrow \texttt{e}_1 \mid \texttt{x}_2 \Rightarrow \texttt{e}_2 \longrightarrow \\ \widehat{\texttt{if}}\ [\texttt{b}]\ \texttt{then}\ \texttt{ite}(\texttt{b},\ \{\widehat{\texttt{v}}/\texttt{x}_1\}\texttt{e}_1,\ \{\widehat{\texttt{v}}_1/\texttt{x}_1\}\texttt{e}_1)\ \texttt{else}\ \texttt{ite}(\texttt{b},\ \{\widehat{\texttt{v}}_2/\texttt{x}_2\}\texttt{e}_2,\ \{\widehat{\texttt{v}}/\texttt{x}_2\}\texttt{e}_2)\end{array}}$$

S-OIf

$$\frac{}{\widehat{\mathcal{E}}[\widehat{\texttt{if}}\ [\texttt{b}]\ \texttt{then}\ \texttt{v}_1\ \texttt{else}\ \texttt{v}_2] \longrightarrow \widehat{\texttt{if}}\ [\texttt{b}]\ \texttt{then}\ \widehat{\mathcal{E}}[\texttt{v}_1]\ \texttt{else}\ \widehat{\mathcal{E}}[\texttt{v}_2]}$$

S-TapeOIf

$$\frac{}{\texttt{tape}\ (\widehat{\texttt{if}}\ [\texttt{b}]\ \texttt{then}\ \texttt{v}_1\ \texttt{else}\ \texttt{v}_2) \longrightarrow \texttt{mux}\ [\texttt{b}]\ (\texttt{tape}\ \texttt{v}_1)\ (\texttt{tape}\ \texttt{v}_2)}$$

S-TapeOVal

$$\frac{\widehat{\texttt{v}}\text{ is oblivious value but not pair}}{\texttt{tape}\ \widehat{\texttt{v}} \longrightarrow \widehat{\texttt{v}}}$$

S-TapePair

$$\frac{}{\texttt{tape}\ (\texttt{v}_1,\ \texttt{v}_2) \longrightarrow (\texttt{tape}\ \texttt{v}_1,\ \texttt{tape}\ \texttt{v}_2)}$$

Weak Values

$$\begin{array}{lcl} \texttt{v} & ::= & \ldots \\ & \mid & \widehat{\texttt{if}}\ [\texttt{b}]\ \texttt{then}\ \texttt{v}\ \texttt{else}\ \texttt{v} \end{array}$$

Evaluation Contexts

$$\begin{array}{lcl} \mathcal{E} & ::= & \ldots \\ & \mid & \widehat{\texttt{if}}\ \square\ \texttt{then}\ \texttt{e}\ \texttt{else}\ \texttt{e} \\ & \mid & \widehat{\texttt{if}}\ \texttt{v}\ \texttt{then}\ \square\ \texttt{else}\ \texttt{e} \\ & \mid & \widehat{\texttt{if}}\ \texttt{v}\ \texttt{then}\ \texttt{v}\ \texttt{else}\ \square \\ & \mid & \texttt{tape}\ \square \end{array}$$

Leaky Contexts

$$\begin{array}{lcl} \widehat{\mathcal{E}} & ::= & \\ & \mid & \square\ \texttt{v} \\ & \mid & \pi_b\ \square \\ & \mid & \texttt{if}\ \square\ \texttt{then}\ \texttt{e}\ \texttt{else}\ \texttt{e} \\ & \mid & \texttt{case}\ \square\ \texttt{of}\ \texttt{x} \Rightarrow \texttt{e} \mid \texttt{x} \Rightarrow \texttt{e} \\ & \mid & \texttt{s}_{\mathbb{B}}\ \square \\ & \mid & \texttt{unfold<X>}\ \square \end{array}$$

Fig. 17. $\lambda_{\text{OADT}\maltese}$ semantics

potential leaks, and is used to ensure that they never occur during execution, as Section 4.2 will discuss in more detail. Finally, $\lambda_{\text{OADT}\maltese}$ updates the syntax for anonymous functions, function types and function definitions with a *leakage label*. A leakage label is either $\top$ or $\bot$, and signals either the presence or the absence of a potential leak, respectively.

4.2 Semantics

The semantics of $\lambda_{\text{OADT}\maltese}$ are an extension of the semantics of λ_{OADT}. Figure 17 shows the new and updated rules; the rest are identical to the rules in Figure 10. This semantics introduces a new syntactic class of *weak values*, which are used to ensure that $\widehat{\texttt{if}}$ does not leak information when evaluated. Weak values simply extend the values in λ_{OADT} with $\widehat{\texttt{if}}$: a $\widehat{\texttt{if}}$ is a weak value if all its subexpressions are weak values. All references to v in the reduction rules (including those not shown in Figure 17) now refer to weak values unless explicitly identified as a value. The semantics also extend evaluation contexts to handle $\widehat{\texttt{if}}$ and tape expressions.

The S-OIf rule captures the key idea of distributing surrounding context into the branches of $\widehat{\texttt{if}}$. Like S-Mux, this rule requires its branches to first be evaluated to weak values using S-Ctx. Note that not all contexts need to be distributed into these branches in order to make progress– pushing fold into $\widehat{\texttt{if}}$ in the expression `fold<tree> (`$\widehat{\texttt{if}}$` [true] ...)` does not gain us anything, for example, since the expression is already a weak value. Figure 17 defines the *leaky contexts* ($\widehat{\mathcal{E}}$) that can be distributed through $\widehat{\texttt{if}}$. For simplicity, we adopt a minimal set of leaky contexts, though allowing more contexts is a potential avenue for optimizing executions. This does not limit the expressivity

of $\lambda_{\text{OADT}\clubsuit}$, for similar reasons to the fold example from above. The semantics of $\widehat{\texttt{case}}$ are also updated to allow potential leaks, with S-OCase now evaluating to $\widehat{\texttt{if}}$ instead of `mux`.

The last three rules in Figure 17 show how to evaluate `tape` annotations. The key idea is to use `tape` as a signal that the context surrounding an $\widehat{\texttt{if}}$ expression has been sufficiently distributed to prevent leaks. Mechanically, whenever a `tape` annotation is applied to $\widehat{\texttt{if}}$ expression whose branches are weak values, it is safe to reduce expression to a secure `mux` using S-TapeOIf. As an example, consider the following expression:

$$\begin{array}{lclclcl}
\texttt{tape}\ (s_{\mathbb{B}}\ (\widehat{\texttt{if}}\ [\texttt{true}] & & \texttt{tape}\ (\widehat{\texttt{if}}\ [\texttt{true}] & & \texttt{mux}\ [\texttt{true}] & & \\
\quad\quad \texttt{then false} & \longrightarrow & \quad \texttt{then}\ (s_{\mathbb{B}}\ \texttt{false}) & \longrightarrow^{*} & \quad (\texttt{tape}\ [\texttt{false}]) & \longrightarrow^{*} & [\texttt{false}] \\
\quad\quad \texttt{else true})) & & \quad \texttt{else}\ (s_{\mathbb{B}}\ \texttt{true})) & & \quad (\texttt{tape}\ [\texttt{true}]) & &
\end{array}$$

After applying S-OIf to distribute the surrounding boolean section $s_{\mathbb{B}}$, the $\widehat{\texttt{if}}$ expression is now annotated with `tape`, and S-TapeOIf can be applied. The `tape` annotations are pushed inside the branches of `mux` to ensure any $\widehat{\texttt{if}}$ expressions they may contain are also repaired. The final two rules ensure `tape` annotations are eventually dropped from oblivious values. An oblivious (non-pair) value annotated with `tape` cannot leak any information, and S-TapeOVal can be applied to remove the extraneous `tape`. S-TapePair allows `tape` annotations to be distributed into the components of an oblivious pair, in order to eventually repair any $\widehat{\texttt{if}}$ expressions they may contain.

4.3 Type System

The typing judgment of $\lambda_{\text{OADT}\clubsuit}$ now includes a leakage label for the typed expression: $\Gamma \vdash e :_{l} \tau$, as do entries in typing contexts Γ. Figure 18 shows a subset of the typing rules of $\lambda_{\text{OADT}\clubsuit}$; the omitted rules are copies of those from λ_{OADT} with straightforward leakage labels annotations. As mentioned in Section 4.1, leakage labels signal whether an expression might contain a potential leak. The reason for these labels is similar to the *security labels* found in other security-type systems [Sabelfeld and Myers 2003; Zdancewic 2002], where type-based information flow control is used to enforce noninterference between high- and low- security information. In $\lambda_{\text{OADT}\clubsuit}$, expressions with $\top$ labels should not influence expressions with $\bot$ labels. In order to minimize the extension to λ_{OADT}, we do not annotate every type with a leakage label, opting to only annotate top-level definitions and function parameters with leakage labels. While it is certainly possible to implement a more precise analysis, this coarse-grained analysis is strong enough for our purposes.

The leakage label of base types is always $\bot$, e.g., in T-Unit. Leakage labels for local or global variables is taken directly from the context, e.g., T-Var. For most public constructs, e.g., T-Pair and T-Proj, the label is the join ($\sqcup$) of the labels of all sub-expressions, where $\bot \sqcup \bot \equiv \bot$ and $\top$ otherwise. T-Proj shows why leakage is an overapproximation, as we cannot always tell which component of a pair labeled with $\top$ is the source of the potential leak. In T-Abs, both the type and label of a parameter are added to the typing context when typing the function body. The label assigned to the function body is then propagated to the whole lambda abstraction. This strategy may seem a bit counterintuitive, as a lambda abstraction is irreducible, and thus cannot leak any information during further evaluation. Of course, while a lambda value will not leak any information on its own, it does have the *potential* to leak when applied to an argument. Because our leakage analysis is quite coarse, we simply consider an expression leaky if it may leak when it is "used". T-App requires a function to be applied to an argument whose label matches that of its parameter. Applying a function with a potentially leaky parameter to a non-leaky argument can be typed by first using the T-Conv rule, which allows the label of an expression to be downgraded. As an example, these rules ensure both $(\lambda x{:}_{\top}\mathbb{B} \Rightarrow s_{\mathbb{B}}\ x)\ (\widehat{\texttt{if}}\ [\texttt{true}]\ \texttt{then true else false})$ and $(\lambda x{:}_{\top}\mathbb{B} \Rightarrow s_{\mathbb{B}}\ x)\ \texttt{true}$ are well-typed expressions.

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

$$\text{T-VAR}\;\frac{x :_l \tau \in \Gamma}{\Gamma \vdash x :_l \tau} \qquad \text{T-UNIT}\;\frac{}{\Gamma \vdash () :_\bot \mathbb{1}} \qquad \text{T-PAIR}\;\frac{\Gamma \vdash e_1 :_{l_1} \tau_1 \quad \Gamma \vdash e_2 :_{l_2} \tau_2 \quad l = l_1 \sqcup l_2}{\Gamma \vdash (e_1,\ e_2) :_l \tau_1 \times \tau_2}$$

$$\text{T-PROJ}\;\frac{\Gamma \vdash e :_l \tau_1 \times \tau_2}{\Gamma \vdash \pi_b\ e :_l \mathtt{ite}(b,\ \tau_1,\ \tau_2)}$$

$$\text{T-ABS}\;\frac{x :_l \tau, \Gamma \vdash e :_{l'} \tau' \qquad \Gamma \vdash \tau :: *}{\Gamma \vdash \lambda x :_l \tau \Rightarrow e :_{l'} \Pi x :_l \tau,\ \tau'} \qquad \text{T-APP}\;\frac{\Gamma \vdash e_1 :_{l_1} \Pi x :_{l_2} \tau_2,\ \tau_1 \qquad \Gamma \vdash e_2 :_{l_2} \tau_2}{\Gamma \vdash e_1\ e_2 :_{l_1} \{e_2/x\}\tau_1}$$

$$\text{T-IF}\;\frac{\Gamma \vdash e_0 :_\bot \mathbb{B} \quad l = l_1 \sqcup l_2 \quad \Gamma \vdash e_1 :_{l_1} \{\mathtt{true}/x\}\tau \quad \Gamma \vdash e_2 :_{l_2} \{\mathtt{false}/x\}\tau}{\Gamma \vdash \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 :_l \{e_0/x\}\tau}$$

$$\text{T-IFNODEP}\;\frac{\Gamma \vdash e_0 :_{l_0} \mathbb{B} \quad l = l_0 \sqcup l_1 \sqcup l_2 \quad \Gamma \vdash e_1 :_{l_1} \tau \quad \Gamma \vdash e_2 :_{l_2} \tau}{\Gamma \vdash \mathtt{if}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 :_l \tau}$$

$$\text{T-CASE}\;\frac{\Gamma \vdash e_0 :_\bot \tau_1 + \tau_2 \quad l = l_1 \sqcup l_2 \quad x_1 :_\bot \tau_1, \Gamma \vdash e_1 :_{l_1} \{\mathtt{inl}{<}\tau_1{+}\tau_2{>}\ x_1/x\}\tau \quad x_2 :_\bot \tau_2, \Gamma \vdash e_2 :_{l_2} \{\mathtt{inr}{<}\tau_1{+}\tau_2{>}\ x_2/x\}\tau}{\Gamma \vdash \mathtt{case}\ e_0\ \mathtt{of}\ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 :_l \{e_0/x\}\tau}$$

$$\text{T-CASENODEP}\;\frac{\Gamma \vdash e_0 :_{l_0} \tau_1 + \tau_2 \quad l = l_0 \sqcup l_1 \sqcup l_2 \quad x_1 :_{l_0} \tau_1, \Gamma \vdash e_1 :_{l_1} \tau \quad x_2 :_{l_0} \tau_2, \Gamma \vdash e_2 :_{l_2} \tau}{\Gamma \vdash \mathtt{case}\ e_0\ \mathtt{of}\ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 :_l \tau}$$

$$\text{T-CONV}\;\frac{\Gamma \vdash e :_{l'} \tau' \quad \tau' \equiv \tau \quad \Gamma \vdash \tau :: * \quad l' \sqsubseteq l}{\Gamma \vdash e :_l \tau}$$

$$\text{T-MUX}\;\frac{\Gamma \vdash e_0 :_\bot \widehat{\mathbb{B}} \quad \Gamma \vdash \tau :: *^O \quad \Gamma \vdash e_1 :_\bot \tau \quad \Gamma \vdash e_2 :_\bot \tau}{\Gamma \vdash \mathtt{mux}\ e_0\ e_1\ e_2 :_\bot \tau} \qquad \text{T-OINJ}\;\frac{\Gamma \vdash e :_\bot \mathtt{ite}(b,\ \tau_1,\ \tau_2) \quad \Gamma \vdash \tau_1 \mathbin{\widehat{+}} \tau_2 :: *^O}{\Gamma \vdash \widehat{\mathtt{in}}_b{<}\tau_1 \mathbin{\widehat{+}} \tau_2{>}\ e :_\bot \tau_1 \mathbin{\widehat{+}} \tau_2}$$

$$\text{T-OIF}\;\frac{\Gamma \vdash e_0 :_\bot \widehat{\mathbb{B}} \quad \Gamma \vdash e_1 :_{l_1} \tau \quad \Gamma \vdash e_2 :_{l_2} \tau}{\Gamma \vdash \widehat{\mathtt{if}}\ e_0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 :_\top \tau}$$

$$\text{T-OCASE}\;\frac{\Gamma \vdash e_0 :_\bot \tau_1 \mathbin{\widehat{+}} \tau_2 \quad x_1 :_\bot \tau_1, \Gamma \vdash e_1 :_{l_1} \tau \quad x_2 :_\bot \tau_2, \Gamma \vdash e_2 :_{l_2} \tau}{\Gamma \vdash \widehat{\mathtt{case}}\ e_0\ \mathtt{of}\ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 :_\top \tau} \qquad \text{T-TAPE}\;\frac{\Gamma \vdash e :_l \tau \quad \Gamma \vdash \tau :: *^O}{\Gamma \vdash \mathtt{tape}\ e :_\bot \tau}$$

Fig. 18. $\lambda_{\text{OADT}✚}$ typing rules

$\lambda_{\text{OADT}✚}$ has dependent and nondependent versions of the typing rule for `if`. In the dependent version, T-IF, the discriminee is not allowed to contain a potential leak, as it may appear in the type. In the nondependent version T-IFNODEP, there is no such restriction, but the type is not allowed to depend on the discriminee. The typing rules for `case`, T-CASE and T-CASENODEP, are similar.

The remaining typing rules deal with expressions that either repair or introduce potential leaks. An expression annotated with `tape` is always assigned the $\bot$ label, as long as that expression has an oblivious type. This is in line with the semantics of `tape`: when applied to an oblivious expression, it eventually evaluates to an oblivious value or a weak value ($\widehat{\mathtt{if}}$). The former is already safe, and the latter can be repaird by S-TAPEOIF. The $\bot$ label in the rule captures the idea that `tape` safely repairs a local leak, such that the surrounding computation can treat it as non-leaky. The rules for $\widehat{\mathtt{if}}$ and $\widehat{\mathtt{case}}$ reflect the fact that they are sources of potential leaks, as both expressions are labeled with $\top$. Both rules require their discriminees to be free of potential leaks, but this does not affect expressiveness, since their discriminees can always be wrapped with `tape`. T-OINJ and T-MUX feature similar requirements.

$\boxed{\Gamma \vdash \tau :: \kappa}$

K-OADT
$$\frac{\texttt{obliv}\ \widehat{X}\ (x{:}\tau) := \tau' \in \Sigma \qquad \Gamma \vdash e :_{\bot} \tau}{\Gamma \vdash \widehat{X}\ e :: *^{O}}$$

K-If
$$\frac{\Gamma \vdash e_0 :_{\bot} \mathbb{B} \qquad \Gamma \vdash \tau_1 :: *^{O} \qquad \Gamma \vdash \tau_2 :: *^{O}}{\Gamma \vdash \texttt{if}\ e_0\ \texttt{then}\ \tau_1\ \texttt{else}\ \tau_2 :: *^{O}}$$

K-Case
$$\frac{\Gamma \vdash e_0 :_{\bot} \tau_1' + \tau_2' \qquad x{:}\tau_1', \Gamma \vdash \tau_1 :: *^{O} \qquad x{:}\tau_2', \Gamma \vdash \tau_2 :: *^{O}}{\Gamma \vdash \texttt{case}\ e_0\ \texttt{of}\ x_1 \Rightarrow \tau_1 \mid x_2 \Rightarrow \tau_2 :: *^{O}}$$

Fig. 19. $\lambda_{\mathsf{OADT}\maltese}$ kinding rules

$\boxed{e \Rrightarrow e'}$

R-OIfCtx
$$\frac{e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2' \qquad \widehat{\mathcal{E}} \Rrightarrow \widehat{\mathcal{E}}'}{\widehat{\mathcal{E}}[\widehat{\texttt{if}}\ [b]\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \Rrightarrow \widehat{\texttt{if}}\ [b]\ \texttt{then}\ \widehat{\mathcal{E}}'[e_1']\ \texttt{else}\ \widehat{\mathcal{E}}'[e_2']}$$

R-TapeOIf
$$\frac{e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2'}{\texttt{tape}\ (\widehat{\texttt{if}}\ [b]\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \Rrightarrow \texttt{mux}\ [b]\ (\texttt{tape}\ e_1')\ (\texttt{tape}\ e_2')}$$

R-TapePair
$$\frac{e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2'}{\texttt{tape}\ (e_1, e_2) \Rrightarrow (\texttt{tape}\ e_1', \texttt{tape}\ e_2')}$$

R-TapeOVal
$$\frac{\widehat{v} \text{ is oblivious value but not pair}}{\texttt{tape}\ \widehat{v} \Rrightarrow \widehat{v}}$$

R-OIf
$$\frac{e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2'}{\widehat{\texttt{if}}\ [b]\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Rrightarrow \texttt{ite}(b, e_1', e_2')}$$

Fig. 20. Subset of $\lambda_{\mathsf{OADT}\maltese}$ parallel reduction rules

Figure 19 shows the updated kinding rules for $\lambda_{\mathsf{OADT}\maltese}$; the other kinding rules are identical to those in Figure 12. The updated rules require types to only depend on terms that do not contain potential leaks, i.e., those assigned the $\bot$ label. To see why, consider the following ill-kinded type:

$\texttt{if}\ (\widehat{\texttt{if}}\ [\texttt{true}]\ \texttt{then}\ \texttt{true}\ \texttt{else}\ \texttt{false})\ \texttt{then}\ \mathbb{1}\ \texttt{else}\ \widehat{\mathbb{B}}$

After distributing the surrounding $\texttt{if}$ into $\widehat{\texttt{if}}$, this reduces to $\widehat{\texttt{if}}\ [\texttt{true}]\ \texttt{then}\ \mathbb{1}\ \texttt{else}\ \widehat{\mathbb{B}}$. Similar expressions at the term level can be repaired by, e.g., distributing $s_{\mathbb{B}}$ through the branches to secure the result of the $\widehat{\texttt{if}}$. At the type level we have no such recourse, however: since types are always public, there is no corresponding way to repair this type by securing its branches.

Figure 20 shows a subset of the updated and new parallel reduction rules. Again, the rules for oblivious constructs are similar to the corresponding step rules. In R-OIfCtx, we write $\widehat{\mathcal{E}} \Rrightarrow \widehat{\mathcal{E}}'$ to mean all the subexpressions in the leaky context take a parallel reduction step. R-OIf is required for confluence, similar to R-Mux. We say a $\lambda_{\mathsf{OADT}\maltese}$ program is well-typed if the global context is well-typed (the updated typing rules for the global context are trivial) and the expression is well-typed with $\bot$ label. The latter restriction ensures that all potential leaks in a $\lambda_{\mathsf{OADT}\maltese}$ program are eventually repaired.

4.4 Type Safety and Obliviousness

The guarantees of the type system of $\lambda_{\text{OADT+}}$ are quite similar to those of λ_{OADT}, although they have been adapted slightly to account for leakage labels. The statement of progress for $\lambda_{\text{OADT+}}$, for example, is limited to expressions without potential leaks:

Theorem 4.1 (Progress). *If* $\cdot \vdash e :_{\bot} \tau$, *then either* $e \longrightarrow e'$ *for some* e', *or* e *is a value. If* $\cdot \vdash \tau :: *^{O}$, *then either* $\tau \longrightarrow \tau'$ *for some* τ', *or* τ *is an oblivious type value.*

This updated statement reflects the fact that leaky expressions only reduce to *weak values*. The proof of this theorem is a consequence of a stronger lemma which also accounts for potentially leaky expressions:

Lemma 4.2. *If* $\cdot \vdash e :_{l} \tau$, *then either* $e \longrightarrow e'$ *for some* e', *or* e *is a weak value.*

The proof of this stronger lemma proceeds similarly to the proof of progress for λ_{OADT}, with the canonical form lemmas extended to weak values. One technicality needed by this proof is a notion of *weak oblivious value*, which extends oblivious values to include $\widehat{\texttt{if}}$ expressions. For the T-Tape case, we have to show that a weak value with an oblivious type is also a weak oblivious value, as `tape` can only be reduced when it is applied to weak oblivious values. This extra lemma requires an updated version of Lemma 3.10, so the proof of progress for $\lambda_{\text{OADT+}}$ now depends on type preservation for parallel reduction. With this lemma in hand, the progress theorem immediately follows from the fact that a weak value is a value if it labeled with $\bot$.

The statements of preservation and obliviousness must also be updated to deal with leakage labels, but are otherwise identical:

Theorem 4.3 (Preservation). *If* $\Gamma \vdash e :_{l} \tau$, *and* $e \longrightarrow e'$, *then* $\Gamma \vdash e' :_{l} \tau$. *If* $\Gamma \vdash \tau :: \kappa$ *and* $\tau \longrightarrow \tau'$, *then* $\Gamma \vdash \tau' :: \kappa$.

Theorem 4.4 (Obliviousness). *If* $e_1 \approx e_2$ *and* $\cdot \vdash e_1 :_{l_1} \tau_1$ *and* $\cdot \vdash e_2 :_{l_2} \tau_2$, *then*

(1) $e_1 \longrightarrow^{n} e_1'$ *if and only if* $e_2 \longrightarrow^{n} e_2'$ *for some* e_2'.
(2) *if* $e_1 \longrightarrow^{n} e_1'$ *and* $e_2 \longrightarrow^{n} e_2'$, *then* $e_1' \approx e_2'$.

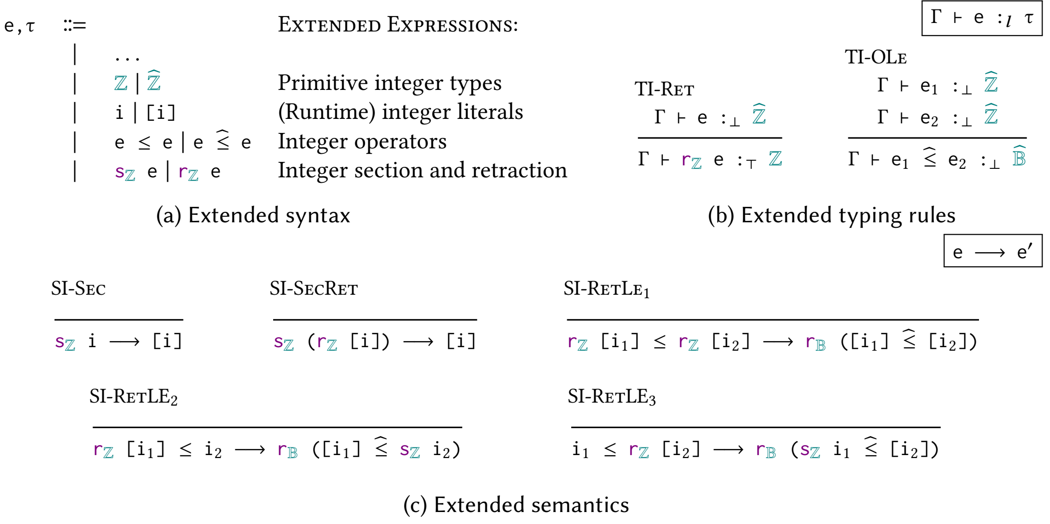
Proofs of both theorems follow the same structure as their counterparts in λ_{OADT}, although many of the lemmas used in the proof of obliviousness now use weak values instead of values.

4.5 Extending $\lambda_{\text{OADT+}}$

This section considers how additional base types might be added to the core calculus of $\lambda_{\text{OADT+}}$, using fixed-width integers as an example. Figure 21 shows a subset of the syntax, semantics and typing rules needed for this new primitive type. The extended language includes public and oblivious versions of integer types, literals, and operators. For simplicity, we only consider a comparison operation, but additional operators could be added in a similar manner. In order to move between the public and oblivious types, section ($s_{\mathbb{Z}}$) and retraction ($r_{\mathbb{Z}}$) operations for integers are also added; both have similar semantics to their boolean counterparts[3]. $r_{\mathbb{Z}}$ always introduces a potential leak, and $r_{\mathbb{Z}}\ \hat{v}$ is considered weak value.

When defining the semantics of potentially leaky expressions like $\leq$, it is important that the semantics does not leak information via the execution trace. When comparing oblivious values with $\leq$, for example, SI-RetLe$_1$ combines $\widehat{\leq}$ and $r_{\mathbb{B}}$ to first securely compare the operands before retracting the resulting oblivious boolean. SI-RetLe$_2$ and SI-RetLe$_3$ are similar, but they apply to cases when one of the operands is not a retraction of an oblivious value by lifting it to oblivious

[3] Although $\lambda_{\text{OADT+}}$ does not include boolean retraction $r_{\mathbb{B}}$ as a primitive, it is easily defined in terms of $\widehat{\texttt{if}}$: $r_{\mathbb{B}}\ e \triangleq \widehat{\texttt{if}}\ e\ \texttt{then true else false}$.



(c) Extended semantics

Fig. 21. A subset of extended language for fixed-width integers

values first. The semantics of other operators can be defined through similar uses of section and retraction functions. As an example, integer addition returns an integer instead of boolean, so we apply $r_{\mathbb{Z}}$ to the result of oblivious addition. If a leaky integer expression is used in a well-typed context, then $r_{\mathbb{Z}}$ will eventually meet $s_{\mathbb{Z}}$ and they can be canceled out via SI-SecRet. Updated versions of evaluation contexts, leaking contexts and the other reduction rules are omitted, as they are straightforward extensions of their counterparts in $\lambda_{\mathsf{OADT}\text{+}}$. The extended typing and kinding rules are also straightforward, and are similar to those for the primitive types in $\lambda_{\mathsf{OADT}\text{+}}$. Figure 21 gives the rules for integer retraction (TI-Ret) and oblivious less-than (TI-OLe) as examples.

4.6 $\lambda_{\mathsf{OADT}\text{+}}$ in action

To demonstrate the expressiveness of $\lambda_{\mathsf{OADT}\text{+}}$, we have written some example oblivious functions and oblivious types with different public views. We have directly encoded these in our Coq development, as well as some accompanying typing and evaluation derivations. All of the examples described in this section are included in our public artifact [Ye and Delaware 2021].

We have encoded the following OADTs for lists and trees. Each oblivious type consists of its type definition, a section function and a retraction function.

- List with the upper bound of its length.
- Tree with the upper bound of its depth.
- Tree with the upper bound of its spine.
- Tree with the upper bound of the number of its vertices (including leaves and nodes).

The second and third of these examples were presented in Section 2. The oblivious tree with the upper bound of its total vertices is the most complicated: while its type definition is effectively an oblivious list, its section and retraction functions correspond to flattening a tree and rebuilding a tree from a list.

In addition to the lookup function from Section 2, we have also written a tree insertion function as a demonstration of how oblivious ADTs are constructed. A more interesting example is a standard map function over oblivious trees, which shows that higher-order functions can be naturally written

in $\lambda_{\text{OADT}✚}$. The following code snippet for an oblivious map function follows the recipe in Figure 6. Label annotations are omitted for brevity, and we use a boolean payload for simplicity.

```
def map̂ (f : 𝔹 → 𝔹) (k : ℕ) (t : treê k) : treê k ≔
  s_tree (map f (r_tree k t)) k
```

The function argument of $\widehat{\texttt{map}}$ takes a public boolean to public boolean, but $\widehat{\texttt{map}}$ could be adapted to accept a function from oblivious boolean to oblivious boolean by composing boolean section and retraction to appropriately "transport" the function argument. The $\widehat{\texttt{map}}$ function could also be adapted any oblivious tree definition by simply replacing $\widehat{\texttt{tree}}$, $\texttt{s}_{\texttt{tree}}$, and $\texttt{r}_{\texttt{tree}}$.

5 RELATED WORK

Secure computation was first formally introduced by Yao [1982] alongside his proposed solution, Garbled Circuits. In secure computation, an untrusted party may observe the whole execution of the secure program, or infer some private information from other side-channels. Enabling secure computations that use algebraic data types that also hide their structures is a key motivation of this work. Secure computation techniques can be broadly divided into those using multiparty computation and those relying on outsourced computation [Evans et al. 2018; Hazay and Lindell 2010]. Those in the former category typically use protocols based on either Garbled Circuits or secret-sharing schemes [Beimel 2011; Goldreich et al. 1987; Maurer 2006]. In the realm of outsourced computation, solutions are typically based on fully homomorphic encryption [Acar et al. 2018; Gentry 2009], but can also be supported by virtualization [Barthe et al. 2014, 2019a] or secure processors [Hoekstra 2015]. These protocols can be used to implement the semantics of λ_{OADT} and $\lambda_{\text{OADT}✚}$.

Many high-level programming languages have been proposed that support some form of secure computation [Hastings et al. 2019]. Their goals are similar to ours in that they provide high-level language support for writing secure programs. However, most do not support (recursive) data structures at all, or assume the structural information is always public. Obliv-C [Zahur and Evans 2015] is a C-like oblivious language. Algebraic data types can be encoded with the C-style `struct` keyword with pointers. Since their oblivious types are restricted to base C types, however, the structure of the defined ADT is public. It would be possible to implement oblivious ADT in Obliv-C by manually padding and using the data types according to their public views. The language provides a `~obliv` keyword that can be used to dynamically track the maximum bound of a data type, at the cost of some additional user effort. Moreover, if the programmers decide to use a different public view, they have to fix every place where this data type is used. ObliVM [Liu et al. 2015] is a Java-like language which also has a `struct` keyword to define data types, but only supports public structures, much like Obliv-C. Wysteria and Wys* [Rastogi et al. 2014, 2019] are functional languages that focus on mixed-mode computation. While they do not support recursive data types, both languages include simple polynomial types and primitive arrays. In contrast, our language does not consider mixed-mode computation. Symphony [Sweet et al. 2021] is a successor of Wysteria which permits more reactive applications through a combination of first-class support for coordinating parties and primitives for secret-sharing and -recombination. Symphony also supports recursive data types which may contain private data, e.g., a tree whose leaves contain oblivious payloads, but does not obfuscate the structure of those datatypes.

Constant-time languages protect programs from inadvertently leaking private information through timing channels by providing atomic constructs and carefully tracking information control. This is also a goal of our system, and our solution to this problem is similar. The first formal study of constant time algorithms was in the context of cache-based attacks [Barthe et al. 2014]. Barthe et al. [2019b] extended the formally verified CompCert compiler [Leroy 2009] to ensure

constant time execution. Though we do not have a compiler for λ_{OADT} or $\lambda_{\text{OADT}\maltese}$, our obliviousness theorem does provide a formal guarantee of a constant-time property. FaCT [Cauligi et al. 2019] is a high-level language for writing constant-time computation using (non-recursive) data types. One of its unique features is a front-end compiler to transform a well-typed (but potentially not constant-time) FaCT program to a constant-time FaCT program. In $\lambda_{\text{OADT}\maltese}$, the programmers can simply encode programs in the conventional fragment and then lift them to oblivious programs (that are constant-time) by composing section and retraction functions.

λ_{obliv} [Darais et al. 2020] is a functional programming language for oblivious computation that focuses on probabilistic programs, making it suitable for implementing some oblivious cryptographic algorithms, such as ORAM, although it does not include algebraic data types. In contrast, our work does not consider probabilistic programs, though it could be an interesting future direction. Our two approaches share similar threat models and guarantees of obliviousness.

Our approach of type-based information flow control to enforce obliviousness, a form of *noninterference*, follows a body of work in *security-type systems* [Sabelfeld and Myers 2003; Zdancewic 2002]. To the best of our knowledge, our system is the first to combine a dependent type system with large elimination and a security-type system. Our notion of retraction bears some resemblance to delimited information release [Sabelfeld and Myers 2004]. In a system with delimited information release, the programmers may choose to reveal some private information, similar to retraction functions in $\lambda_{\text{OADT}\maltese}$. However, our semantics guarantees retraction never releases any private information. Another difference with a standard security-type system is that we use explicit coercion via section functions instead of implicit subtyping to convert public types to secure types. On the one hand, our typing rules and semantics for oblivious types and non-oblivious types are quite different. On the other hand, implicit subtyping does not make sense in the case of ADTs. To convert a public ADT to an oblivious one, we not only need to know how the oblivious ADT is represented, but also to infer the public views.

Our oblivious types can also be viewed as a kind of refinement types [Kawaguchi et al. 2009; Rondon et al. 2008; Xi and Pfenning 1999]: the oblivious tree in our running example can be understood as trees with a maximum depth stipulated by the type index, for example. However, this declarative specification does not explain how to represent such an oblivious tree. Nonetheless, this view of subset types suggests a future direction of integrating refinement typing into our system to ensure the correct use of public indices. Dependent type systems with large elimination can be found in many theorem provers, such as Coq [The Coq Development Team 2021] and Agda [Norell [n.d.]]. These languages are designed more towards theorem proving and thus only admit total functions, while our languages allow general recursion and hence nontermination. A notable dependently typed languages with nontermination is Zombie [Sjöberg 2015; Sjöberg et al. 2012; Sjöberg and Weirich 2015], though our goals are drastically different.

Nanevski et al. [2013] show how Relational Hoare Type Theory can be used to encode and verify a variety of security policies in a theorem prover using dependent types. While capable of specifying security policies like noninterference, their encoding does not address termination behavior and only characterizes the final output value, and thus does not protect against control flow leaks. In addition, users have to manually verify these properties in the proof assistant. In contrast, we consider a much stronger threat model, and both our oblivious calculi protect against a larger class of leaks. Both calculi additionally provide a fixed security guarantee in the form of our obliviousness theorem, which any well-typed program enjoys “for free”, without any additional user effort.

In our mechanized formalization, the correctness and security guarantees provided by the underlying cryptographic primitives are baked into our semantics and notion of indistinguishability. There is a body of work about formally verified cryptography [Abate et al. 2021; Barthe et al. 2011,

2009], which could be integrated into our work in the future to provide a stronger formal guarantee. Some of these solutions have focused on verifying multiparty computation [Backes et al. 2010; Haagh et al. 2018].

Another popular cryptographic technique for hiding private information of data structures is oblivious RAM [Goldreich 1987; Goldreich and Ostrovsky 1996; Stefanov et al. 2013] (ORAM). ORAM provides primitives to access an encrypted memory buffer without revealing the access pattern, except for the number of accesses. There have been proposals for generically constructing oblivious data structures using ORAM [Wang et al. 2014]. Oblivious data structures constructed this way hide the access patterns of a sequence of data structure operations. This line of work in general does not consider leakage through side-channels. While our solution also naturally hides access patterns, we also assume a much stronger adversary who can observe the whole computation.

6 DISCUSSION AND FUTURE WORK

In contrast to our "dynamic" approach of repairing potential leakage in $\lambda_{\text{OADT}+}$, it is also possible to repair programs statically, at least for some simple programs. The key observation is that we can apply fusion techniques [Ohori and Sasano 2007; Wadler 1990] to implementations like the one in Figure 8. By fusing the composition of section, public function and retraction, it is possible to build an implementation from oblivious input to oblivious output, without the intermediate public trees generated by the retraction function. While this method works for some simple examples like an append function for lists, only a limited amount of functions can be fully fused. Nonetheless, we believe this is a direction worth exploring in the future.

One natural next step is the implementation of an algorithmic type checker, although the possibility of nontermination in our languages poses a potential challenge to dependent type checking. One solution is to simply ask the programmers to provide a "fuel" to bound how many steps the type checker can take, similar to Zombie [Sjöberg et al. 2012]. We could also only allow total functions at the type level. As dependent types only occur in section and retraction functions, which have rather specialized forms, we believe a weaker and incomplete type checker would work in practice. Building an implementation of $\lambda_{\text{OADT}+}$ is also an important piece of future work, where performance may play a critical role. While oblivious programs are generally quite slow in practice due to timing channel protections, our unconventional semantics may introduce additional overheads.

Other future directions include the formalization of a general algorithm for synthesizing implementations like Figure 8, synthesizing section and retraction functions, inference of leakage labels, and automatic insertion of `tape` annotations.

7 CONCLUSION

To our best knowledge, this work is the first programming language that supports hiding the structure of rich recursive data types in secure computations. We have presented λ_{OADT}, a core calculus for encoding oblivious programs over oblivious algebraic data types. λ_{OADT} combines dependent types with large elimination to represent oblivious algebraic data types, and provides a security-type system to ensure that computations reveal no private information over what is provided by the public view of the data. To enable programmers to write a single function and easily build secure programs with different public views, we have also developed $\lambda_{\text{OADT}+}$. This language is equipped with a novel semantics that repairs potential leaks without compromising the security guarantees of λ_{OADT}. We have proved, mechanically, that our solution provides a strong and formal security guarantee: an adversary can not infer any private information, even given the entire execution trace of a program.

ACKNOWLEDGMENTS

We thank Robert Dickerson, Pedro Abreu, Aaron Stump, and the anonymous reviewers for their detailed comments and suggestions. We also thank Kirshanthan Sundararajah, Milind Kulkarni, Chaitanya Koparkar, Michael Vollmer and Ryan Newton for their stimulating discussions. This material is based upon work partially supported by the National Science Foundation under Grant CCF-1755880, the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA) under contract #2019-19020700004, and the Purdue Graduate School under a Summer Research Grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, ODNI, IARPA, or Purdue. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

REFERENCES

Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–15. https://doi.org/10.1109/CSF51468.2021.00048

Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2018. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *ACM Computing Surveys (CSUR)* 51, 4 (July 2018), 79:1–79:35. https://doi.org/10.1145/3214303

Michael Backes, Matteo Maffei, and Esfandiar Mohammadi. 2010. Computationally Sound Abstraction and Verification of Secure Multi-Party Computations. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 8)*, Kamal Lodaya and Meena Mahajan (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 352–363. https://doi.org/10.4230/LIPIcs.FSTTCS.2010.352

Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-Level Non-Interference for Constant-Time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, Scottsdale, Arizona, USA, 1267–1279. https://doi.org/10.1145/2660267.2660283

Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. 2019a. System-Level Non-Interference of Constant-Time Cryptography. Part I: Model. *Journal of Automated Reasoning* 63, 1 (June 2019), 1–51. https://doi.org/10.1007/s10817-017-9441-5

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2019b. Formal Verification of a Constant-Time Preserving C Compiler. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 7:1–7:30. https://doi.org/10.1145/3371075

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology – CRYPTO 2011 (Lecture Notes in Computer Science)*, Phillip Rogaway (Ed.). Springer Berlin Heidelberg, 71–90.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 90–101. https://doi.org/10.1145/1480881.1480894

Amos Beimel. 2011. Secret-Sharing Schemes: A Survey. In *Coding and Cryptology (Lecture Notes in Computer Science)*, Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing (Eds.). Springer, Berlin, Heidelberg, 11–46. https://doi.org/10.1007/978-3-642-20901-7_2

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for Timing-Sensitive Computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, Phoenix, AZ, USA, 174–189. https://doi.org/10.1145/3314221.3314605

David Darais, Ian Sweet, Chang Liu, and Michael Hicks. 2020. A Language for Probabilistically Oblivious Computation. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–31. https://doi.org/10.1145/3371118 arXiv:1711.09305

David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Foundations and Trends® in Privacy and Security* 2, 2-3 (2018), 70–246. https://doi.org/10.1561/3300000019

Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing (STOC '09)*. Association for Computing Machinery, New York, NY, USA, 169–178.

https://doi.org/10.1145/1536414.1536440

J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. https://doi.org/10.1109/SP.1982.10014

O. Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. Association for Computing Machinery, New York, NY, USA, 182–194. https://doi.org/10.1145/28395.28416

O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. Association for Computing Machinery, New York, New York, USA, 218–229. https://doi.org/10.1145/28395.28420

Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (May 1996), 431–473. https://doi.org/10.1145/233551.233553

Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. 2018. Computer-Aided Proofs for Multiparty Computation with Active Security. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 119–131. https://doi.org/10.1109/CSF.2018.00016

M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 479–496. https://doi.org/10.1109/SP.2019.00028

Carmit Hazay and Yehuda Lindell. 2010. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer, Berlin ; London.

Matthew E Hoekstra. 2015. Intel SGX for Dummies (Intel SGX Design Objectives). https://www.intel.com/content/www/us/en/develop/blogs/protecting-application-secrets-with-intel-sgx.html

Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. 2009. Type-Based Data Structure Verification. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, Dublin, Ireland, 304–315. https://doi.org/10.1145/1542476.1542510

Peeter Laud and Liina Kamm (Eds.). 2015. *Applications of Secure Multiparty Computation*. Number volume 13 in Cryptology and Information Security Series. IOS Press, Amsterdam, Netherlands.

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107. https://doi.org/10.1145/1538788.1538814

C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. 359–376. https://doi.org/10.1109/SP.2015.29

Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - a Secure Two-Party Computation System. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, USA, 20.

Ueli Maurer. 2006. Secure Multi-Party Computation Made Simple. *Discrete Applied Mathematics* 154, 2 (Feb. 2006), 370–381. https://doi.org/10.1016/j.dam.2005.03.020

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2013. Dependent Type Theory for Verification of Information Flow and Access Control Policies. *ACM Transactions on Programming Languages and Systems* 35, 2 (July 2013), 6:1–6:41. https://doi.org/10.1145/2491522.2491523

Ulf Norell. [n.d.]. The Agda Wiki. https://wiki.portal.chalmers.se/agda/Main/HomePage

Atsushi Ohori and Isao Sasano. 2007. Lightweight Fusion by Fixed Point Promotion. *ACM SIGPLAN Notices* 42, 1 (Jan. 2007), 143–154. https://doi.org/10.1145/1190215.1190241

A. Rastogi, M. A. Hammer, and M. Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *2014 IEEE Symposium on Security and Privacy*. 655–670. https://doi.org/10.1109/SP.2014.48

Aseem Rastogi, Nikhil Swamy, and Michael Hicks. 2019. Wys*: A DSL for Verified Secure Multi-Party Computations. In *Principles of Security and Trust (Lecture Notes in Computer Science)*, Flemming Nielson and David Sands (Eds.). Springer International Publishing, 99–122. https://doi.org/10.1007/978-3-030-17138-4_5

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

A. Sabelfeld and A.C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Andrei Sabelfeld and Andrew C. Myers. 2004. A Model for Delimited Information Release. In *Software Security - Theories and Systems (Lecture Notes in Computer Science)*, Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki (Eds.). Springer Berlin Heidelberg, 174–191. https://doi.org/10.1007/978-3-540-37621-7_9

Vilhelm Sjöberg. 2015. A Dependently Typed Language with Nontermination. *Publicly Accessible Penn Dissertations* (Jan. 2015). https://repository.upenn.edu/edissertations/1137

Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. 2012. Irrelevance, Heterogeneous Equality, and Call-by-Value Dependent Type

Systems. *Electronic Proceedings in Theoretical Computer Science* 76 (Feb. 2012), 112–162. https://doi.org/10.4204/EPTCS.76.9 arXiv:1202.2923

Vilhelm Sjöberg and Stephanie Weirich. 2015. Programming up to Congruence. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 369–382. https://doi.org/10.1145/2676726.2676974

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 *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/2508859.2516660

Ian Sweet, David Darais, David Heath, Ryan Estes, William Harris, and Michael Hicks. 2021. Symphony: A Concise Language Model for MPC. In *Informal Proceedings of the Workshop on Foundations on Computer Secuirty (FCS)*.

The Coq Development Team. 2021. The Coq Proof Assistant. (Jan. 2021). https://doi.org/10.5281/zenodo.4501022

Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science* 73, 2 (June 1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, Scottsdale, Arizona, USA, 215–226. https://doi.org/10.1145/2660267.2660314

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/292540.292560

Andrew C. Yao. 1982. Protocols for Secure Computations. In *23rd Annual Symposium on Foundations of Computer Science (Sfcs 1982)*. 160–164. https://doi.org/10.1109/SFCS.1982.38

Qianchuan Ye and Benjamin Delaware. 2021. Oblivious Algebraic Data Types: POPL22 Artifact. Zenodo. https://doi.org/10.5281/zenodo.5652106

Samee Zahur and David Evans. 2015. *Obliv-C: A Language for Extensible Data-Oblivious Computation.* Technical Report 1153. https://eprint.iacr.org/2015/1153

Stephan Arthur Zdancewic. 2002. *Programming Languages for Information Security.* Ph.D. Dissertation. Cornell University, USA.