CERIAS Tech Report 2024-3
Language-Based Techniques for Policy-Agnostic Oblivious Computation
by Qianchuan Ye
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

LANGUAGE-BASED TECHNIQUES FOR POLICY-AGNOSTIC OBLIVIOUS COMPUTATION

by

Qianchuan Ye

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

May 2024

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Benjamin Delaware, Chair

Department of Computer Science

Dr. Roopsha Samanta

Department of Computer Science

Dr. Milind Kulkarni

Elmore Family School of Electrical and Computer Engineering

Dr. David Darais

Galois, Inc.

Dr. Tianyi Zhang

Department of Computer Science

Approved by:

Dr. Kihong Park

To my loving mother

ACKNOWLEDGMENTS

I would not have survived my PhD career without the immense help and support from many humans and cats.

First of all, I thank my advisor Benjamin Delaware. I started my PhD program with zero background in any of the topics that I enjoy doing now. Ben taught me everything from doing research in general to solving a specific Coq error. I still remember the good old days when we sat down together and he patiently guided me through my Coq proofs. He is always willing to answer my dumb questions and listen to my rants. Ben provided me more than enough support and encouragement while giving me freedom to pursue whatever I was interested in at the time. His insights, questions, advice and sense of humor made me a better researcher, writer, presenter and a better person. I am forever in debt to him.

I also thank my PhD advisory committee. The "policy-agnostic" direction, a key component of this dissertation, is due to Roopsha Samanta's suggestions when we worked on the HACCLE project together. Her "Reasoning about Programs" course is also my favorite class during my PhD. Milind Kulkarni's sharp ideas, questions and comments shaped this dissertation and my presentations, as well as pointed out promising future directions. David Darais gave a lot of insightful feedback to this dissertation, and his work on oblivious languages has been a source of inspiration. The helpful feedback from my examining committee, Christina Garman and Tianyi Zhang, also improved this dissertation. I would like to thank other professors whom I have worked with or received valuable advice from: Suresh Jagannathan, Xiaokang Qiu, Jenna DiVincenzo, and Vasileios Zikas.

I have learned a lot from my now and former colleagues, Pedro Abreu, Rob Dickerson, Patrick LaFontaine, Prasita Mukherjee, Zhe Zhou, Kia Rahmani, Eric Bond and Paul Krogmeier. I have had many stimulating discussions with Raghav Malik and Kirshanthan Sundararajah. Their ideas and suggestions have been made into this dissertation. It was also delightful to study category theory with Raghav, Rob and Zhe.

Many thanks to my friends at PurPL group, including Guannan Wei, Chris Wagner, Nouraldin Jaber, Yongwei Yuan, Ashish Mishra, Pratyush Das, Adhitha Dias, Artem Pelenitsyn, Julia Belyakova, Oliver Bračevac, Durga Keerthi Mandarapu, Vidhush Singhal and

Vani Nagarajan. I have enjoyed the talks and conversations in our PurPL seminars (and free lunch). I thank all my friends at Purdue, including Guanhong Tao, Jianliang Wu, Le Yu, Yiran Hu, Yuyan Bao, Shengwei An, Zikang Xiong, Yilin Zheng, Chuyang Ke, Habiba Farrukh, Seunghoon Lee and Basavesh Ammanaghatta Shivakumar. They have made my PhD life at Purdue less boring.

I am grateful to my dear friend Jianfei Gao. He is a wonderful workout and show-watching buddy and an amazing chef, although I would prefer less garlic. His cats, Π and Ξ, deserve my special thanks for making me less productive and disrupting my sleep during the two years we spent together. Π is the sweetest lady in the world who loves sleeping on my lap when I need to get work done. She single-handedly converted me into a cat person. Ξ is, well, a little asshole.

I have met some wonderful people outside Purdue in various summer schools and conferences. Kuen-Bang Hou (Favonia) and Yao Li gave me much helpful advice. Natarajan Shankar has many fun stories to tell, and the conversations with him are always inspiring. I feel fortunate to have had the opportunity to work with Clément Pit-Claudel and Adam Chlipala. Ian Sweet and his work have inspired this dissertation, especially the implementations. Ashish Kundu has given valuable feedback to this work and helped me along the way. My friends from summer schools, Matthew Yacavone, Ramana Nagasamudram, Ziyang Li, Yiyun Liu, Jacob Prinz and Mako Bates, made these summer schools a much more enjoyable experience. Vikraman Choudhury helped me understand adjunction better. Loïc Pujet recommended me some materials about cubical type theory that I still have not read; one day I will go down this rabbit hole.

Finally, I want to thank my beloved mom and brother. They have been extremely supportive not just during my PhD, but throughout my life. Without them I would not even make it to a PhD program.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Protecting personal information is growing increasingly important to the general public, to the point that major tech companies now advertise the privacy features of their products. Despite this, it remains challenging to implement applications that do not leak private information either directly or indirectly, through timing behavior, memory access patterns, or control flow side channels. Existing security and cryptographic techniques such as secure multiparty computation (MPC) provide solutions to privacy-preserving computation, but they can be difficult to use for non-experts and even experts.

This dissertation develops the design, theory and implementation of various language-based techniques that help programmers write privacy-critical applications under a strong threat model. The proposed languages support private structured data, such as trees, that may hide their structural information and complex policies that go beyond whether a particular field of a record is private. More crucially, the approaches described in this dissertation decouple privacy and programmatic concerns, allowing programmers to implement privacy-preserving applications modularly, i.e., to independently develop application logic and independently update and audit privacy policies. Secure-by-construction applications are derived automatically by combining a standard program with a separately specified security policy.

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 in the early 1980s by Yao [1], along with one of the first examples, Yao's Millionaires' Problem. In this problem, Alice and Bob are millionaires, and they wish to know who is richer without disclosing their wealth. In other words, this is a two-party secure computation that calculates a boolean result $x \leq y$ from private integer inputs x and y provided by the two parties, without revealing these two integers. Starting from this simple problem, secure computation has since found many privacy-focused applications, including secure auctions, voting, and privacy-preserving machine learning [2–4].

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 [2]. MPC is typically implemented using cryptography-based protocols, such as Yao's Garbled Circuits [1] or secret-sharing [5, 6], while outsourced computation can be implemented using a variety of mechanisms, including cryptography-based fully homomorphic encryption [7, 8], virtualization [9, 10] and secure processors [11].

Writing secure applications that directly use these techniques can be quite challenging and error-prone, however, even if the author has the requisite cryptographic expertise. Thus, several high-level programming languages and compilers have been created to help programmers write secure applications, starting with Fairplay, the first publicly available MPC compiler [12]. For example, Obliv-C [13] is a C-like language for MPC applications which compiles down to Yao's Garbled Circuits. Other notable languages include PICCO [14], ObliVM [15], Wysteria/Wys* [16, 17], λ_{obliv} [18], Viaduct [19], and Symphony [20]. Chapter 7 discusses these languages in greater detail.

1.1 Problem Description

While these languages raise the level of abstraction, it remains challenging to develop complex secure applications in them, due to their lack of or limited support for rich data structures, complex policies on data, and the separation of programmatic and privacy concerns.

Consider a private decision tree classification problem. Suppose Alice owns a medical record, represented as a *record* data type with several fields, such as her ID, age, height and weight. On the other hand, Bob provides a decision tree, each of whose nodes compares a feature (i.e., field) from Alice's record against a threshold, and whose leaves are decisions. The computation is a standard decision tree classification algorithm that traverses Bob's decision tree according to the comparison result of each node, until it reaches a decision. While it is straightforward to write this simple application in a conventional programming language, it is challenging to implement a secure version in the aforementioned languages.

To begin with, many of these languages 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. What if Bob wants to hide also the structure of his decision tree, so that observing this data structure itself or how it is used does not disclose whether the tree is left-heavy or right-heavy? Unfortunately, in general it is impossible to hide everything about a recursive data: some public information has to be disclosed to bound the data's in-memory representation and the computation over this data, e.g., recursion depth; the secure computation may simply not terminate otherwise. If Bob wishes to hide his tree's structure, he at a minimum needs to reveal its maximum depth, for example. An important question for supporting secure data structures is thus what information about a private data may be publicly shared. Each choice of publicly disclosed information, or *public view*, defines a *privacy policy* on the data.

One major obstacle to securely implementing applications that respect their policies is the possibility of *timing channels*: as one example, 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 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* [9] is easy to implement for computations over simple data types like arrays, it becomes more complicated for richer data structures, like the decision tree in our motivating example. First, users have to decide how to manually "pad" data structures, so that they are consistent with a particular public view. Second, programmers also have to track the public view throughout the program, making sure it remains consistent throughout. These manual efforts essentially force the programmers to explicitly enforce the privacy policies within the logic of the application itself; as a consequence, the entire application must be examined in order to audit its privacy policy. Lastly, richer data structures can have multiple public views, i.e., privacy policies, representing different trade-offs between privacy and performance. An application must be rewritten for each of these policies, due to the intermixing of policy enforcement and application logic.

This is particularly true for applications with the sorts of complex requirements that can occur in practice. Within the United States, for example, the Health Insurance Portability and Accountability Act (HIPAA) governs how patient data may be used. HIPAA allows *either* the personally identifiable information (PII) *or* medical data to be shared, but not both. Notably, this policy does not simply specify whether some particular field of a patient's

medical record is private or public; rather it is a *relation* that dictates how a program can access and manipulate different parts of every individual record. To conform to this policy, a secure application must either pay the (considerable) cryptographic overhead of conservatively securing all accesses to the fields of a record, or adopt a more sophisticated strategy for monitoring how data is accessed. These challenges become more acute when dealing with recursive data, e.g., lists or trees, whose policies are necessarily more complex. In the private decision tree classification problem, if Bob, the owner of the tree, stipulates that only its depth may be disclosed, the classification function must use secure operations to ensure that no other information about the tree is leaked, e.g., its spine or the attributes it uses. If Bob is willing to share the latter bits of information, however, this function must either be rewritten to take advantage of the new, more permissive policy, or continue to pay the cost of providing stricter privacy guarantees. Thus, the intermixing of privacy and programmatic concerns in current languages require users to write different implementations of essentially the same program for each distinct privacy policy, and makes it difficult to read, write, and reason about secure applications. Ideally, these concerns should be separated, allowing programmers to write the functionality of their program once and for all, and then select the right public view for their security and performance requirements.

This dissertation considers the following research questions.

- Can we design a secure language that supports rich data structures that may hide their own structures? Can this language support complex policies on data, e.g., the either-or policy for medical records?

- How can we guarantee that no sensitive information is leaked when executing programs in this language, even if an attacker can observe the structured data itself and how it is manipulated? To faithfully model the domain of secure multiparty computation, we need to consider a strong threat model where a powerful attacker can observe every intermediate state of program executions, which also naturally covers timing side-channels.

- How do we decouple privacy policies from application logic? This form of modularity allows users to implement applications and specify policies independently.

1.2 Contributions and Outline

This dissertation develops the design, theory and implementation of various novel programming language techniques for *oblivious computation*. For generality, this dissertation uses the term oblivious computation to mean computation that does not leak private information directly or indirectly, e.g., through side-channels, under a strong threat model. The solutions described in this dissertation can be applied to any oblivious computation, including secure multiparty computation, fully homomorphic encryption, and other secure computation techniques that are not based on cryptography.

These solutions provide a foundation to a functional programming language that is rich, safe, and accessible.

- **Rich**. Using our language, programmers can implement oblivious applications that involve rich data structures with complex policies, and use other high-level functional programming features such as higher-order functions.

- **Safe**. No attacker can infer any private information beyond their own private input and the publicly shared information, even when they are able to observe the data representation and every single program state in an execution of a program written in this language.

- **Accessible**. Programmers can write the application logic in the standard way, without knowing the underlying secure computation model and the particular privacy policies used for the applications. In other words, the language is *policy-agnostic*. Privacy policies are separately defined, specified and audited. The functionality and privacy policies can then be composed to derive the secure implementation. This modular design allows functionality and policies to be reusable and makes it easy to switch policies for, e.g., making tradeoffs between privacy and performance.

The remainder of this dissertation is structured as follows: Chapter 2 introduces the important concepts and ideas that underline or inspire the work in this dissertation, e.g., *noninterference* [21]. This chapter develops a simple security-typed language and a simple dependently typed language, to illustrate the formalization and proof techniques for these

systems, which are the key inspiration for the dissertation. The example core calculi in Chapter 2 are designed to be simple while similar in style to the novel systems described in subsequent chapters.

The next four chapters discuss the main technical contributions of this dissertation.

- Chapter 3 describes *oblivious algebraic data types* (OADTs), a form of dependent types that can encode complex privacy policies for structured data. This chapter also develops $\lambda_{\texttt{OADT}}$, a core calculus for writing oblivious programs using OADTs, and describes the strong security guarantees provided by its type system.

- Chapter 4 tackles the problem of decoupling privacy and programmatic concerns. This chapter presents a dynamic approach for enforcing privacy policies automatically, using a novel operational semantics called *tape semantics*. This semantics allows programs to include unsafe computations, and then repairs these unsafe computations at runtime, which is the key to modularizing these concerns. A core calculus extending $\lambda_{\texttt{OADT}}$, dubbed $\lambda_{\texttt{OADT+}}$, is presented, and various theoretic guarantees are established.

- Chapter 5 discusses the implementation of a policy-agnostic programming language based on $\lambda_{\texttt{OADT+}}$, called Taype. This chapter presents a bidirectional type checker and a compilation pipeline that addresses various challenges in implementing OADTs and tape semantics. The chapter concludes with an evaluation of this language.

- Chapter 6 proposes a static approach for automatic policy enforcement. This approach transforms source programs into secure target programs that respect the given privacy policies, enabling significant performance improvements over previous dynamic approach, i.e., tape semantics, and better policy specifications. The Taypsi language described in this chapter, and its underlying core calculus $\lambda_{\texttt{OADT}\Psi}$, still provide the same security guarantee and separation of privacy concerns and application logic.

Finally, related work is discussed in Chapter 7, and Chapter 8 summarizes the dissertation and proposes future directions.

The material in this dissertation is mainly based on Ye and Delaware [22–24]. The formalization of all the core calculi presented in this dissertation and the proofs of their

metatheory have been mechanized in the Coq proof assistant. Chapter 3, Chapter 4, Chapter 5 and Chapter 6 include references to the publicly available artifacts of each of these formal developments, as well as an implementation of the languages.

2. BACKGROUND

Before describing the technical contributions of this dissertation, we begin with an introduction to several important concepts and established techniques.

2.1 Threat Models

This dissertation adopts a semi-honest setting [25]. In other words, the attackers are passive, in that they follow the computation or protocols faithfully, but may try to infer secrets from the information they can gather while doing so. Although they do not act maliciously, e.g., executing the programs in a deliberately wrong way, private information can still be leaked inadvertently. An attacker may be able to observe some extra information besides the input and output that they are permitted to see, e.g., how long a program runs, and infer sensitive information from such *side-channels* indirectly. One major challenge for oblivious computation is to protect against leakage through these side-channels.

The strength of a threat model in the semi-honest setting is determined by the capability of an attacker, i.e., what information they can observe. Consider a standard imperative programming model, where computations can be thought of as state transformations. Each state is a store, or memory, containing data, indexed by variable names. Some portion of this memory may be protected, and can only be accessed by trusted entities. A variable pointing to values in this secret section of memory is tagged with a *security label* $\top$ (high-security), indicating that the contents of this variable are not visible to any attackers, the low-security observers. On the other hand, variables labeled with $\bot$ (low-security) have values that are visible to everyone, including attackers. A secure program in this programming model should ensure no high-security data values can be inferred from low-security data or other information available via side-channels. For example, assigning a high-security variable `h` to a low-security variable `l`, `l := h`, breaks the security guarantee, as an attacker can obtain the secret `h` by observing the value of `l` after the assignment. A secure programming language should reject this program as insecure.

Undesirable information flow from high-security to low-security can happen implicitly as well. The following example modifies the low-security variable `l` according to whether the secret `h` is greater than 0.

```
if h > 0 then l := 1 else l := 0
```

By observing `l`, an attacker can infer some information about the secret `h`, which should be prohibited by the system.

An attacker may be able to observe more than just the low-security portion of the memory. For example, they may observe the running time of a program. This sort of timing behavior can also reveal secrets, as illustrated by the following example.

```
if h > 0 then h := 1 else (delay 10; h := 0)
```

Even though this program does not modify any low-security variables, an attacker can still infer some information about the initial value of `h` and thus obtain the final assignment to `h`, by measuring how much time this program takes to finish. A secure system needs to make sure high-security information does not influence the timing behavior of a program, if an attacker can gather information from such *timing channels*.

Perhaps surprisingly, the following innocent-looking program is also insecure if executed under a standard semantics.

```
if h > 0 then h := 1 else h := 0
```

While this program indeed reveals no private information through output and timing channels, a powerful attacker is still able to peek into `h` by observing which branch this program takes via *control flow channels*.

Our final strong threat model reflects both those of standard MPC protocols based on simultaneous execution of the programs, e.g., secret-sharing [6, 26], and those based on outsourced computation where untrusted evaluators perform the execution, e.g., fully homomorphic encryption [7]. In these protocols, *any* party involved in the computation could be an attacker, including the ones executing the programs, forcing us to protect against a powerful attacker that can observe the whole execution, including every intermediate program state. As a result, we have to obscure which branches the program takes, for example. This threat model also naturally covers weaker adversaries, including those who can only observe

the timing behavior. The language solutions developed in this thesis are designed to be secure under this strong threat model.

2.2 Security Specifications

One standard notion for specifying information security is *noninterference* [21], which roughly states that protected data can not influence observable information, e.g., low-security data. Hence, an observer cannot infer any protected information by observing low-security information. There are many variants of noninterference [27], reflecting different semantics and threat models. This property is usually defined relationally: for example, given *indistinguishable* inputs that differ only in their protected components, a program always produces outputs that are also *indistinguishable*. Intuitively, this means if the high-security values in the input change, the observable information in the execution remains unaffected, even when the high-security portion of the output can vary.

Indistinguishability specifies what information an observer can see, and is usually defined using *security labels*. For example, a high-security label ($\top$) may correspond to confidentiality, which describes private information, while a low-security label ($\bot$) describes public information. In the multiparty setting, principal identifiers such as `Alice` and `Bob` may be used as security labels. Alice may only have access to her own data, i.e., information labeled with `Alice`, for instance. Indistinguishability is then an equivalence relation ($\approx$) indexed by observers, stating that the observable information, according to its security labels, in the data is the same. In a system with high and low labels, for example, data are indistinguishable to an attacker if their low-security information is the same.

Another main component in a definition of noninterference are the outputs of a computation, i.e., what information is available to an observer in an execution. If we are only concerned about the computed result, then its output is simply the result. However, a program's output can include information in side-channels as well. For example, we may define the output of a program as the result it computes to *and* its running time, if we wish to account for timing behavior.

The general recipe for defining noninterference is thus to specify an indistinguishability relation on a program's data and the output of a program, according to its semantics and threat model. Noninterference can then be formalized as when indistinguishable program states (configurations) produce indistinguishable outputs.

Example 2.2.1 (Termination-insensitive noninterference)**.** Under a standard imperative semantics, a program configuration is a pair (P, σ), where P is a program and σ is a map from variable names to their values, i.e., machine state, tagged by a $\top$ or $\bot$ label. A program's behavior can be described by a big-step operational semantics $P, \sigma \Downarrow \sigma'$, meaning that the program P, when run in initial state σ, evaluates to final state σ'.

Consider a threat model where attackers can only see the values of variables labeled with $\bot$. In this threat model, we can define indistinguishability on a pair of states as those in which all the variables labelled with $\bot$ have the same values:

$$\sigma \approx \sigma' \triangleq \sigma_\bot = \sigma'_\bot$$

where $\sigma_\bot$ means the $\bot$ projection of σ, consisting of all the variables that have the $\bot$ label. The output is simply the final state. We can then define *termination-insensitive* noninterference as follows: given any initial states σ_1 and σ_2 such that $\sigma_1 \approx \sigma_2$, if a program P with these states evaluates to final states σ'_1 and σ'_2, i.e., $P, \sigma_1 \Downarrow \sigma'_1$ and $P, \sigma_2 \Downarrow \sigma'_2$, then $\sigma'_1 \approx \sigma'_2$. Note that this definition is termination-insensitive because we assume P terminates with σ and σ', otherwise this property is vacuously satisfied.

Alternatively, we can fit this definition into the previous framing more directly by lifting the indistinguishability notion to program configurations: $P, \sigma \approx P', \sigma'$ if and only if $P = P'$ and $\sigma \approx \sigma'$. We can also extend the notion of output to include a divergence symbol which indicates (P, σ) does not terminate (assuming this language does not have other diverging behaviors such as exceptions). We can then say two outputs are indistinguishable if one of them is the divergence symbol or they are indistinguishable states. With these generalized definitions, noninterference is simply that indistinguishable program configurations generate

indistinguishable outputs. Choosing to instead only relate the divergence symbol to itself leads to *termination-sensitive* noninterference:

Example 2.2.2 (Termination-sensitive noninterference)**.** Consider an attacker who can also observe termination behavior. Noninterference is defined as follows: given any initial states σ_1 and σ_2 such that $\sigma_1 \approx \sigma_2$, a program P either diverges under both initial states, or evaluates to a pair of final states σ_1' and σ_2', i.e., $P, \sigma_1 \Downarrow \sigma_1'$ and $P, \sigma_2 \Downarrow \sigma_2'$, such that $\sigma_1' \approx \sigma_2'$.

Example 2.2.3 (Probabilistic noninterference)**.** Under a nondeterministic and probabilistic computation model, program semantics may be defined as $P, \sigma \Downarrow D$, where D is a probability distribution on states. To protect against attackers who can sample multiple runs, we may consider two outputs indistinguishable if the generated distributions are statistically indistinguishable.

Example 2.2.4 (Timing channels)**.** As illustrated in Section 2.1, timing behavior can be used to infer private information. The big-step semantics we used previously is too coarse for this threat model. While it is possible to define a big-step semantics that also calculates the running time compositionally, it is more straightforward to define a small-step semantics $(P, \sigma) \longrightarrow (P', \sigma')$, which means a program P with state σ takes one computation step to the configuration (P', σ'). The program P in a configuration can be thought of as a program counter. For example, the program `(x := x + 1; y := 2)` with state $\{x \mapsto 0\}$ steps to `(y := 2)` with state $\{x \mapsto 1\}$.

With this small-step semantics, noninterference can be defined as follows: given initial states σ_1 and σ_2, if $(P, \sigma_1) \longrightarrow^{n_1} (\epsilon, \sigma_1')$ and $(P, \sigma_2) \longrightarrow^{n_2} (\epsilon, \sigma_2')$, then $\sigma_1' \approx \sigma_2'$ and $n_1 = n_2$, where ϵ is the empty program. The judgment $(P, \sigma) \longrightarrow^{n} (P', \sigma')$ generalizes the previous single-step version, meaning that (P, σ) takes n steps to (P', σ'). This noninterference definition counts how many steps it takes for a program to finish in order to ensure the number of steps taken is independent of the secrets. In this formulation, we assume each step costs the same amount of time for simplicity, but this can be easily adapted to capture more precise timing behavior, e.g., by annotating the small-step semantics with a more precise cost for each command. This version of the definition also does not consider termination channels, but it is easy to state a variant similar to termination-sensitive noninterference.

Example 2.2.5 (Access patterns)**.** Attackers in some threat models can observe memory access patterns. To capture this capability, we may extend the previous small-step semantics to also record memory access events $(P, \sigma) \xrightarrow{\texttt{ev}} (P', \sigma')$. For example, the program (`x := read l; ...`) may step to (...) with the new state $\{x \mapsto 0\}$ and the generated event `READ(l, ⊤, 0)`. This read event records the memory location `l`, its security label and the read result 0.

The output of an execution is a list of the generated events under this small-step semantics. Indistinguishability of events and the noninterference definition can be straightforwardly formulated similarly to the previous attempts. One variant of this kind of noninterference is *memory trace obliviousness* [28].

Example 2.2.6 (Control flow channels)**.** In a strong threat model, an attacker may be able to inspect the program counter in the CPU and the executing instructions. This is the case in many oblivious computations, including MPC, as discussed in Section 2.1; as any party executing the programs can be thought of as an attacker who has full control over their own computing device. To model a powerful attacker who can observe every program state, we consider output to be the (possibly infinite) traces of configurations under a small-step semantics. In other words, the output is the (possibly infinite) list $(P_1, \sigma_1); (P_2, \sigma_2); \ldots$ if the execution is $(P_1, \sigma_1) \longrightarrow (P_2, \sigma_2) \longrightarrow \ldots$. Indistinguishability of traces is then simply pair-wise indistinguishability of program configurations.

Going back to the last example in Section 2.1, the following two traces are distinguishable, despite the indistinguishable initial program configurations.

$$(\texttt{if h > 0 then h := 1 else h := 0}, \{h \mapsto 1\}) \longrightarrow (\texttt{h := 1}, \{h \mapsto 1\}) \longrightarrow (\epsilon, \{h \mapsto 1\})$$

$$(\texttt{if h > 0 then h := 1 else h := 0}, \{h \mapsto 0\}) \longrightarrow (\texttt{h := 0}, \{h \mapsto 0\}) \longrightarrow (\epsilon, \{h \mapsto 0\})$$

Observe that an attacker can infer the value of `h` by observing the second configuration in these traces, i.e., the branch it takes, and this program does not satisfy noninterference.

$$
\begin{array}{rcll}
\mathtt{e} & ::= & \mathtt{b}_l \mid \mathtt{x} \mid \lambda \mathtt{x}{:}\tau{\Rightarrow}\mathtt{e} \mid \mathtt{e}\ \mathtt{e} \mid \mathtt{if}\ \mathtt{e}\ \mathtt{then}\ \mathtt{e}\ \mathtt{else}\ \mathtt{e} & \textsc{Expressions} \\
\tau & ::= & \mathbb{B}_l \mid \tau{\to}\tau & \textsc{Types} \\
l & ::= & \top \mid \bot & \textsc{Labels} \\
\mathtt{v} & ::= & \mathtt{b}_l \mid \mathtt{x} \mid \lambda \mathtt{x}{:}\tau{\Rightarrow}\mathtt{e} & \textsc{Values}
\end{array}
$$

Figure 2.1. Syntax of a simple security-typed calculus

$\boxed{\mathtt{e} \Downarrow \mathtt{v}}$

$$
\textsc{E-Val}\ \frac{}{\mathtt{v} \Downarrow \mathtt{v}}
\qquad
\textsc{E-App}\ \frac{\mathtt{e}_1 \Downarrow \lambda \mathtt{x}{:}\tau{\Rightarrow}\mathtt{e}'_1 \qquad \mathtt{e}_2 \Downarrow \mathtt{v}_2 \qquad [\mathtt{v}_2/\mathtt{x}]\mathtt{e}'_1 \Downarrow \mathtt{v}}{\mathtt{e}_1\ \mathtt{e}_2 \Downarrow \mathtt{v}}
$$

$$
\textsc{E-IfTrue}\ \frac{\mathtt{e}_0 \Downarrow \mathtt{true}_l \qquad \mathtt{e}_1 \Downarrow \mathtt{v}}{\mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \Downarrow \mathtt{v}}
\qquad
\textsc{E-IfFalse}\ \frac{\mathtt{e}_0 \Downarrow \mathtt{false}_l \qquad \mathtt{e}_2 \Downarrow \mathtt{v}}{\mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \Downarrow \mathtt{v}}
$$

Figure 2.2. Semantics of a simple security-typed calculus

2.3 Security-Type Systems

Various language properties can be ensured statically by a well-designed type system. Security-typed languages [27, 29] have been extensively studied to enforce information-flow properties such as noninterference. This section presents a functional language as a small extension to simply typed lambda calculus (STLC) with high- and low-clearance booleans. This core calculus is designed to be minimal in order to exhibit the key idea of using a type system to restrict how certain resources can be used, while remaining similar in style to the proposed calculi in this dissertation.

Figure 2.1 shows the syntax of this core calculus. Terms and types in this language are mostly standard. Boolean literals and boolean types are annotated with a security label, indicating a boolean value is only visible by observers with enough permission according to its label.

This language has a standard (big-step) operational semantics, as shown in Figure 2.2. Note that the labels are ignored at runtime (and hence can be erased), and the language relies solely on its type system for the security guarantees. This is different from variants

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

$$\textsc{T-Lit}\ \frac{}{\Gamma \vdash \mathtt{b}_l : \mathbb{B}_l} \qquad \textsc{T-Var}\ \frac{\mathtt{x} : \tau \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau} \qquad \textsc{T-Abs}\ \frac{\mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e} : \tau_2}{\Gamma \vdash \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e} : \tau_1 \to \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash \mathtt{e}_2 : \tau_1 \to \tau_2 \qquad \Gamma \vdash \mathtt{e}_1 : \tau_1}{\Gamma \vdash \mathtt{e}_2\ \ \mathtt{e}_1 : \tau_2}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathbb{B}_l \qquad \Gamma \vdash \mathtt{e}_1 : \tau \qquad \Gamma \vdash \mathtt{e}_2 : \tau \qquad l \sqsubseteq \mathtt{label}(\tau)}{\Gamma \vdash \mathtt{if}\ \ \mathtt{e}_0\ \ \mathtt{then}\ \ \mathtt{e}_1\ \ \mathtt{else}\ \ \mathtt{e}_2 : \tau}$$

Figure 2.3. Typing rules of a simple security-typed calculus

of security-typed languages that may implicitly *promote* labels to ones of higher security, effectively "moving" low-security values to a more protected memory region.

The key idea of a security-type system is to rule out the programs with undesirable information flow by analyzing the security level of each component and imposing restrictions accordingly. A type system that enforces these restrictions for our simple security-typed language is presented in Figure 2.3. Most of the typing rules are standard. Boolean literals are ascribed a boolean type with the same label in T-Lit. T-If is the most interesting rule. This rule uses the expected label comparison operator $\sqsubseteq$ (e.g., $\bot \sqsubseteq \top$), and the meta-function `label` which obtains a type's security label: $\mathtt{label}(\mathbb{B}_l) = l$ and $\mathtt{label}(\tau_1 \to \tau_2) = \bot$. Note that for simplicity we assume functions are always visible to attackers (hence the $\bot$ label). While it is possible to assign a high-security label to functions [29], functions are generally public information in secure multiparty computation and many other oblivious computations. The side condition in T-If enforces a crucial policy that an `if`-conditional's branches can only be "more secure" than its condition, otherwise we risk revealing the value of the condition from the result of a conditional. For example, the expression $\mathtt{if}\ \mathtt{true}_\top\ \mathtt{then}\ \mathtt{true}_\bot\ \mathtt{else}\ \mathtt{false}_\bot$ is ill-typed, since a low-observer can see the computed result $\mathtt{true}_\bot$ and infer that the condition is also `true`.

The goal of this type system is to guarantee noninterference. Unlike imperative languages, this pure functional language is stateless, and its program configurations are simply expressions. However, establishing noninterference is not necessarily easier, due to the existence of higher-

$$\boxed{\mathtt{e} \approx \mathtt{e}'}$$

$$\frac{}{\mathtt{b}_\bot \approx \mathtt{b}_\bot} \qquad \frac{}{\mathtt{b}_\top \approx \mathtt{b}'_\top} \qquad \frac{}{\mathtt{x} \approx \mathtt{x}} \qquad \frac{\mathtt{e} \approx \mathtt{e}'}{\lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{e} \approx \lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{e}'} \qquad \frac{\mathtt{e}_1 \approx \mathtt{e}'_1 \qquad \mathtt{e}_2 \approx \mathtt{e}'_2}{\mathtt{e}_1 \;\; \mathtt{e}_2 \approx \mathtt{e}'_1 \;\; \mathtt{e}'_2}$$

$$\frac{\mathtt{e}_0 \approx \mathtt{e}'_0 \qquad \mathtt{e}_1 \approx \mathtt{e}'_1 \qquad \mathtt{e}_2 \approx \mathtt{e}'_2}{\mathtt{if} \;\; \mathtt{e}_0 \;\; \mathtt{then} \;\; \mathtt{e}_1 \;\; \mathtt{else} \;\; \mathtt{e}_2 \approx \mathtt{if} \;\; \mathtt{e}'_0 \;\; \mathtt{then} \;\; \mathtt{e}'_1 \;\; \mathtt{else} \;\; \mathtt{e}'_2}$$

Figure 2.4. Indistinguishability of expressions in a simple security-typed language

order functions, i.e., lambda abstractions, even if they are low-security. Formally, we want to prove the following noninterference theorem.

Theorem 2.3.1 (Noninterference)**.** *If* $\mathtt{x} : \mathbb{B}_\top \vdash \mathtt{e} : \mathbb{B}_\bot$ *and* $\cdot \vdash \mathtt{v}_1, \mathtt{v}_2 : \mathbb{B}_\top$*, then* $[\mathtt{v}_1/\mathtt{x}]\mathtt{e} \Downarrow \mathtt{v} \iff [\mathtt{v}_2/\mathtt{x}]\mathtt{e} \Downarrow \mathtt{v}$.

This definition is similar to Zdancewic [29, Theorem 3.1.1]. It says the (low-clearance) result can not be affected by a high-clearance value (i.e., secure boolean) in an expression.

This theorem is simply a special case of Lemma 2.3.5 below. It can also be straightforwardly generalized to indistinguishable expressions and indistinguishable substitutions, but these stronger noninterference theorems are also corollaries of Lemma 2.3.5, or can be achieved using the same proof techniques. A direct proof of these noninterference theorems (including Theorem 2.3.1) will not succeed, however, similar to the challenge in proving normalization of STLC: reduction of a lambda application may result in bigger terms that "escape" the induction hypotheses.

To formally establish noninterference, we first define indistinguishability of expressions as inference rules in Figure 2.4. The first two rules establish that two high-security booleans are indistinguishable, and all other rules are simply congruence rules. Intuitively, two expressions are indistinguishable if they only differ in their unobservable, secure boolean values. Note that attackers are allowed to peek under the binder of a lambda abstraction, as it is the case under the strong threat model used later in this dissertation. This definition of indistinguishability for lambda abstractions is not strong enough to prove noninterference directly, because substituting indistinguishable arguments in indistinguishable function bodies may result in

bigger terms that are not obviously indistinguishable after evaluation. To strengthen this equivalence relation, we extend indistinguishability to the stronger *logical relation* [30, 31] below.

Definition 2.3.1. This relation is mutually defined via a pair of set-valued type denotations: a value interpretation $\mathcal{V}[\![\tau]\!]$ and an expression interpretation $\mathcal{E}[\![\tau]\!]$. We say closed and well-typed terms $\mathtt{e}$ and $\mathtt{e}'$ are equivalent at type τ if $(\mathtt{e}, \mathtt{e}') \in \mathcal{E}[\![\tau]\!]$. In other words, they evaluate to equivalent values, per $\mathcal{V}[\![\tau]\!]$. Importantly, the desirable property of substitution is directly encoded in the value interpretation of function type. In addition, the logical relation is naturally extended to typing contexts $\mathcal{G}[\![\Gamma]\!]$, in order to relate equivalent substitutions. We use the notation $\sigma \vdash \Gamma$ to mean that $\cdot \vdash \sigma(\mathtt{x}) : \tau$ for every $\mathtt{x} : \tau \in \Gamma$, similar to the well-typedness side conditions in other interpretations.

$$\mathcal{V}[\![\mathbb{B}_\bot]\!] = \{\, (\mathtt{b}_\bot, \mathtt{b}_\bot) \,\} \qquad\qquad \mathcal{V}[\![\mathbb{B}_\top]\!] = \{\, (\mathtt{b}_\top, \mathtt{b}'_\top) \,\}$$

$$\mathcal{V}[\![\tau_1 \to \tau_2]\!] = \left\{ (\lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e}, \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e}') \;\middle|\; \begin{array}{l} \cdot \vdash \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e} : \tau_1 \to \tau_2 \wedge \cdot \vdash \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e}' : \tau_1 \to \tau_2 \wedge \\ \mathtt{e} \approx \mathtt{e}' \wedge \\ \forall (\mathtt{v}, \mathtt{v}') \in \mathcal{V}[\![\tau_1]\!].([\mathtt{v}/\mathtt{x}]\mathtt{e}, [\mathtt{v}'/\mathtt{x}]\mathtt{e}') \in \mathcal{E}[\![\tau_2]\!] \end{array} \right\}$$

$$\mathcal{E}[\![\tau]\!] = \{\, (\mathtt{e}, \mathtt{e}') \mid \cdot \vdash \mathtt{e} : \tau \wedge \cdot \vdash \mathtt{e}' : \tau \wedge \mathtt{e} \Downarrow \mathtt{v} \wedge \mathtt{e}' \Downarrow \mathtt{v}' \wedge (\mathtt{v}, \mathtt{v}') \in \mathcal{V}[\![\tau]\!] \,\}$$

$$\mathcal{G}[\![\Gamma]\!] = \{\, (\sigma, \sigma') \mid \sigma \vdash \Gamma \wedge \sigma' \vdash \Gamma \wedge \forall \mathtt{x} : \tau \in \Gamma.(\sigma(\mathtt{x}), \sigma'(\mathtt{x})) \in \mathcal{V}[\![\tau]\!] \,\}$$

The base cases of the denotation of values can also be defined directly using our indistinguishability relation. For example:

$$\mathcal{V}[\![\mathbb{B}_\bot]\!] = \{\, (\mathtt{v}, \mathtt{v}') \mid \cdot \vdash \mathtt{v} : \mathbb{B}_\bot \wedge \cdot \vdash \mathtt{v}' : \mathbb{B}_\bot \wedge \mathtt{v} \approx \mathtt{v}' \,\}$$

This style is equivalent because a well-typed value has a canonical form that is determined by its type.

Lemma 2.3.2 (Canonical forms)**.**

- *If* $\cdot \vdash \mathtt{v} : \mathbb{B}_\ell$*, then* $\mathtt{v} = \mathtt{true}_\ell$ *or* $\mathtt{v} = \mathtt{false}_\ell$*.*

- *If* $\cdot \vdash \mathtt{v} : \tau_1 \to \tau_2$, *then* $\mathtt{v} = \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e}$ *for some* $\mathtt{e}$.

Proof. Straightforward case analysis on the type derivation. □

A standard substitution lemma is needed to discharge the well-typedness side conditions in the logical relations.

Lemma 2.3.3 (Substitution preserves typing relation)**.** *If* $\Gamma \vdash \mathtt{e} : \tau$ *and* $\sigma \vdash \Gamma$, *then* $\cdot \vdash \sigma(\mathtt{e}) : \tau$.

Proof. Routine induction on the typing derivation. □

We also need another substitution lemma about indistinguishability. Indistinguishability is naturally extended to substitutions pointwise: $\sigma \approx \sigma'$ if $\sigma(\mathtt{x}) \approx \sigma'(\mathtt{x})$ for each $\mathtt{x}$ in their domain.

Lemma 2.3.4 (Substitution preserves indistinguishability)**.** *If* $\sigma \approx \sigma'$, *then* $\sigma(\mathtt{e}) \approx \sigma'(\mathtt{e})$.

Proof. Routine induction on the structure of $\mathtt{e}$. □

Now we can state and prove a general lemma that substitution preserves the logical relations.

Lemma 2.3.5 (Substitution preserves logical relation)**.** *If* $\Gamma \vdash \mathtt{e} : \tau$ *and* $(\sigma, \sigma') \in \mathcal{G}[\![\Gamma]\!]$, *then* $(\sigma(\mathtt{e}), \sigma'(\mathtt{e})) \in \mathcal{E}[\![\tau]\!]$.

Proof. By induction on the derivation of the typing judgment. The cases T-Lit and T-Var are trivial. In the rest of the proof, we do not explicitly prove the well-typedness side conditions, since they are simply consequences of Lemma 2.3.3.

Case T-Abs: We need to show $(\lambda \mathtt{x}{:}\tau_1 \Rightarrow \sigma(\mathtt{e}), \lambda \mathtt{x}{:}\tau_1 \Rightarrow \sigma'(\mathtt{e})) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$. First, $\sigma(\mathtt{e}) \approx \sigma'(\mathtt{e})$ follows from Lemma 2.3.4 and the fact that $\sigma \approx \sigma'$ when $(\sigma, \sigma') \in \mathcal{G}[\![\Gamma]\!]$. Next, suppose $(\mathtt{v}, \mathtt{v}') \in \mathcal{V}[\![\tau_1]\!]$, we have $(\sigma[\mathtt{x} \mapsto \mathtt{v}], \sigma'[\mathtt{x} \mapsto \mathtt{v}']) \in \mathcal{G}[\![\mathtt{x} : \tau_1, \Gamma]\!]$. It then follows from the induction hypothesis that $(\sigma[\mathtt{x} \mapsto \mathtt{v}](\mathtt{e}), \sigma'[\mathtt{x} \mapsto \mathtt{v}'](\mathtt{e})) \in \mathcal{E}[\![\tau_2]\!]$. That is $([\mathtt{v}/\mathtt{x}]\sigma(\mathtt{e}), [\mathtt{v}'/\mathtt{x}]\sigma'(\mathtt{e})) \in \mathcal{E}[\![\tau_2]\!]$, as required.

Case T-App: We have by the induction hypotheses:

- $(\sigma(\mathtt{e}_2), \sigma'(\mathtt{e}_2)) \in \mathcal{E}[\![\tau_1 \to \tau_2]\!]$
- $(\sigma(\mathtt{e}_1), \sigma'(\mathtt{e}_1)) \in \mathcal{E}[\![\tau_1]\!]$

Therefore:

- $\sigma(\mathtt{e}_2) \Downarrow \lambda \mathtt{x}\!:\!\tau_1 \Rightarrow \mathtt{u}_2$ for some $\mathtt{u}_2$
- $\sigma'(\mathtt{e}_2) \Downarrow \lambda \mathtt{x}\!:\!\tau_1 \Rightarrow \mathtt{u}'_2$ for some $\mathtt{u}'_2$
- $(\lambda \mathtt{x}\!:\!\tau_1 \Rightarrow \mathtt{u}_2, \lambda \mathtt{x}\!:\!\tau_1 \Rightarrow \mathtt{u}'_2) \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$
- $\sigma(\mathtt{e}_1) \Downarrow \mathtt{v}_1$ for some $\mathtt{v}_1$
- $\sigma'(\mathtt{e}_1) \Downarrow \mathtt{v}'_1$ for some $\mathtt{v}'_1$
- $(\mathtt{v}_1, \mathtt{v}'_1) \in \mathcal{V}[\![\tau_1]\!]$

The value interpretation of $\tau_1 \to \tau_2$ allows us to derive that $([\mathtt{v}_1/\mathtt{x}]\mathtt{u}_2, [\mathtt{v}'_1/\mathtt{x}]\mathtt{u}'_2) \in \mathcal{E}[\![\tau_2]\!]$, which again gives us:

- $[\mathtt{v}_1/\mathtt{x}]\mathtt{u}_2 \Downarrow \mathtt{v}$ for some $\mathtt{v}$
- $[\mathtt{v}'_1/\mathtt{x}]\mathtt{u}'_2 \Downarrow \mathtt{v}'$ for some $\mathtt{v}'$
- $(\mathtt{v}, \mathtt{v}') \in \mathcal{V}[\![\tau_2]\!]$

We then have $\sigma(\mathtt{e}_2)\ \ \sigma(\mathtt{e}_1) \Downarrow \mathtt{v}$ and $\sigma'(\mathtt{e}_2)\ \ \sigma'(\mathtt{e}_1) \Downarrow \mathtt{v}'$ from E-App with the desired property $(\mathtt{v}, \mathtt{v}') \in \mathcal{V}[\![\tau_2]\!]$. That is $(\sigma(\mathtt{e}_2)\ \ \sigma(\mathtt{e}_1), \sigma'(\mathtt{e}_2)\ \ \sigma'(\mathtt{e}_1)) \in \mathcal{E}[\![\tau_2]\!]$.

Case T-If: We want to show:

$$(\mathtt{if}\ \ \sigma(\mathtt{e}_0)\ \ \mathtt{then}\ \ \sigma(\mathtt{e}_1)\ \ \mathtt{else}\ \ \sigma(\mathtt{e}_2), \mathtt{if}\ \ \sigma'(\mathtt{e}_0)\ \ \mathtt{then}\ \ \sigma'(\mathtt{e}_1)\ \ \mathtt{else}\ \ \sigma'(\mathtt{e}_2)) \in \mathcal{E}[\![\tau]\!]$$

By the induction hypothesis, we know $\sigma(\mathtt{e}_0) \Downarrow \mathtt{b}_l$ and $\sigma'(\mathtt{e}_0) \Downarrow \mathtt{b}'_l$ for some $\mathtt{b}$ and $\mathtt{b}'$, such that $(\mathtt{b}_l, \mathtt{b}'_l) \in \mathcal{V}[\![\mathbb{B}_l]\!]$. We consider two cases.

If $l = \bot$, then $\mathtt{b} = \mathtt{b}'$. Without loss of generality, assume they equal `true` (the case of `false` is the similar). It suffices to show $\sigma(\mathtt{e}_1) \Downarrow \mathtt{v}$ and $\sigma'(\mathtt{e}_1) \Downarrow \mathtt{v}'$ for some $\mathtt{v}$ and $\mathtt{v}'$ such that $(\mathtt{v}, \mathtt{v}') \in \mathcal{V}[\![\tau]\!]$. But it follows immediately from the induction hypothesis.

If $l = \top$, then $\texttt{label}(\tau) = \top$, which means τ is $\mathbb{B}_\top$. By the induction hypotheses, we know all 4 branches evaluate to some values, and they are (secure) boolean literals by Lemma 2.3.2. Since $(\mathtt{b}_\top, \mathtt{b}'_\top) \in \mathcal{E}[\![\mathbb{B}_\top]\!]$ for any $\mathtt{b}$ and $\mathtt{b}'$, the proof is concluded trivially. □

Finally, Theorem 2.3.1, i.e., noninterference, is a direct consequence of Lemma 2.3.5.

2.4 Dependent Type Systems

Dependently typed languages [32] allow types to depend on terms, providing strong static guarantees about behavior of programs. For example, the type of lists may depend on a natural number which specifies the length of a list. Full-spectrum dependent type systems form the foundation of many theorem provers, such as Coq [33], Lean [34], and Agda [35]. These languages support *large elimination*, or computing types from data. Another form of dependent types is refinement types [36–39]. These systems usually do not support large elimination, but rather augment the types with predicates that restrict the values to a "subset" of the type. Refinement types are designed to be lighter-weight, and many program specifications can be expressed through these refinement types. Unlike Coq or Agda, where proofs are constructed inside the language based on the *Curry-Howard Correspondence*, refinement type systems rely heavily on automated theorem provers, e.g., Z3 [40], to discharge proof obligations, trading expressiveness for more automation. There are also works that aim to bring dependent types to general purpose languages [41–43].

Dependently typed languages, especially those with large elimination, directly inspired the novel type systems introduced in this dissertation. This section presents a simple dependently typed language that extends the simply typed lambda calculus with dependent functions and dependent conditionals. While this core calculus is similar in spirit to pure type systems (PTS) [32] in the lambda cube [44, 45], it is closer in style to λLF [46, Chapter 2.2] and the type systems in this dissertation.

Figure 2.5 shows the syntax of this simple dependently typed language. Similar to most dependently typed languages, types and terms belong to the same syntactic class. By convention, we try to use the metavariable τ for types and $\mathtt{e}$ to refer to terms. The key extension to simply typed lambda calculus is the inclusion of dependent function types

EXPRESSIONS

$$e, \tau \quad ::= \quad b \mid x \mid \lambda x{:}\tau \Rightarrow e \mid e\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \mathbb{B} \mid \Pi x{:}\tau,\tau$$

Figure 2.5. Syntax of a simple dependently typed calculus

$\boxed{e \Rrightarrow e'}$

$$\text{R-Refl}\ \frac{}{e \Rrightarrow e} \qquad \text{R-App}\ \frac{e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2'}{(\lambda x{:}\tau \Rightarrow e_2)\ e_1 \Rrightarrow [e_1'/x]e_2'} \qquad \text{R-IfTrue}\ \frac{e_1 \Rrightarrow e_1'}{\texttt{if true then } e_1 \texttt{ else } e_2 \Rrightarrow e_1'}$$

$$\text{R-IfFalse}\ \frac{e_2 \Rrightarrow e_2'}{\texttt{if false then } e_1 \texttt{ else } e_2 \Rrightarrow e_2'} \qquad \text{R-AbsCgr}\ \frac{\tau \Rrightarrow \tau' \qquad e \Rrightarrow e'}{\lambda x{:}\tau \Rightarrow e \Rrightarrow \lambda x{:}\tau' \Rightarrow e'} \qquad \text{R-AppCgr}\ \frac{e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2'}{e_1\ e_2 \Rrightarrow e_1'\ e_2'}$$

$$\text{R-IfCgr}\ \frac{e_0 \Rrightarrow e_0' \qquad e_1 \Rrightarrow e_1' \qquad e_2 \Rrightarrow e_2'}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \Rrightarrow \texttt{if } e_0' \texttt{ then } e_1' \texttt{ else } e_2'} \qquad \text{R-PiCgr}\ \frac{\tau_1 \Rrightarrow \tau_1' \qquad \tau_2 \Rrightarrow \tau_2'}{\Pi x{:}\tau_1,\tau_2 \Rrightarrow \Pi \tau_1',\tau_2'}$$

Figure 2.6. Parallel reduction of a simple dependently typed calculus

(Π) and dependent conditionals (`if`) which allow for type-level computation. For example, $\Pi x{:}\mathbb{B},\texttt{if } x \texttt{ then } \mathbb{B} \texttt{ else } \mathbb{B} \to \mathbb{B}$ is a valid type: the return type of this function is dictated by its argument `x`. We write $\tau_1 \to \tau_2$ for $\Pi x{:}\tau_1,\tau_2$ when `x` does not appear in τ_2, indicating that there is no dependency in this type.

While we can give this language a standard call-by-value or call-by-name small-step operational semantics, we need a set of more permissive reduction rules to establish the desired metatheoretic properties. A standard technique is to define an alternative semantics called *parallel reduction*, shown in Figure 2.6. These semantics rules allow reductions under the binder of a lambda abstraction (R-AbsCgr) and reductions in `if` branches (R-IfCgr). All the subcomponents of an expression are reduced simultaneously in the congruence rules and even in the β-reduction rules (R-App). As type-level computation is supported, reductions can also happen in types (e.g., R-PiCgr). Note that the standard call-by-value or call-by-name reduction relations are included in this "bigger" parallel reduction relation, so properties such as preservation (i.e., subject reduction) also hold for these more restricted semantics.

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

$$\textsc{T-Lit}\ \frac{}{\Gamma \vdash \mathtt{b} : \mathbb{B}} \qquad \textsc{T-Var}\ \frac{\mathtt{x} : \tau \in \Gamma \qquad \Gamma \vdash \tau :: *}{\Gamma \vdash \mathtt{x} : \tau} \qquad \textsc{T-Abs}\ \frac{\mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e} : \tau_2 \qquad \Gamma \vdash \tau_1 :: *}{\Gamma \vdash \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e} : \Pi \mathtt{x}{:}\tau_1, \tau_2}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash \mathtt{e}_2 : \Pi \mathtt{x}{:}\tau_1, \tau_2 \qquad \Gamma \vdash \mathtt{e}_1 : \tau_1}{\Gamma \vdash \mathtt{e}_2\ \ \mathtt{e}_1 : [\mathtt{e}_1/\mathtt{x}]\tau_2}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathbb{B} \qquad \Gamma \vdash \mathtt{e}_1 : [\mathtt{true}/\mathtt{z}]\tau \qquad \Gamma \vdash \mathtt{e}_2 : [\mathtt{false}/\mathtt{z}]\tau \qquad \Gamma \vdash [\mathtt{e}_0/\mathtt{z}]\tau :: *}{\Gamma \vdash \mathtt{if}\ \ \mathtt{e}_0\ \ \mathtt{then}\ \ \mathtt{e}_1\ \ \mathtt{else}\ \ \mathtt{e}_2 : [\mathtt{e}_0/\mathtt{z}]\tau}$$

$$\textsc{T-Conv}\ \frac{\Gamma \vdash \mathtt{e} : \tau \qquad \tau \equiv \tau' \qquad \Gamma \vdash \tau' :: *}{\Gamma \vdash \mathtt{e} : \tau'}$$

Figure 2.7. Typing rules of a simple dependently typed calculus

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

$$\textsc{K-Bool}\ \frac{}{\Gamma \vdash \mathbb{B} :: *} \qquad \textsc{K-Pi}\ \frac{\Gamma \vdash \tau_1 :: * \qquad \mathtt{x} : \tau_1, \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \Pi \mathtt{x}{:}\tau_1, \tau_2 :: *} \qquad \textsc{K-If}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathbb{B} \qquad \Gamma \vdash \tau_1 :: * \qquad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \mathtt{if}\ \ \mathtt{e}_0\ \ \mathtt{then}\ \ \tau_1\ \ \mathtt{else}\ \ \tau_2 :: *}$$

Figure 2.8. Kinding rules of a simple dependently typed calculus

Programs in this language are typed using a pair of typing and kinding judgments, $\Gamma \vdash \mathtt{e} : \tau$ and $\Gamma \vdash \tau :: *$ respectively. Hence this is a 2-layer type system: terms have types, and types have kinds, although there is only one kind $*$ to keep this language simple. Figure 2.7 and Figure 2.8 show the (mutually defined) typing and kinding rules.

The kinding rules are mostly straightforward, with the most interesting rule K-If allowing for large elimination on booleans. Some typing rules (T-Var, T-Abs and T-If) have side conditions about kinding to ensure the types used in these rules are well-formed. T-Abs is similar to the standard typing rule for lambda abstraction, although τ_2 is allowed to refer to the variable being bound. T-App is also similar to its standard counterpart, but the return type of the function is specialized with its argument after the application. Typing a

dependent conditional (T-If) relies on an implicit *motive* that is specialized when typing its branches, because the overall type of the expression may depend on its condition. This motive (τ) contains a special free variable ($\mathtt{z}$) which stands in for the result of the condition. This variable is concretized with `true` ($[\mathtt{true}/\mathtt{z}]\tau$) when typing the `then` branch, for example. Finally, T-Conv allows any well-typed term to be typed with an equivalent type, using a type equivalence relation $\tau \equiv \tau'$. This equivalence is defined directly in terms of the parallel reduction relation. Two terms are said to be equivalent if they can parallelly reduce to the same term in zero or more steps:

$$\tau_1 \equiv \tau_2 \triangleq \exists \tau.\ \tau_1 \Rrightarrow^* \tau \wedge \tau_2 \Rrightarrow^* \tau$$

It is also possible to define this equivalence inductively using parallel reduction, or define it as the transitive and symmetric closure of parallel reduction. These definitions are equivalent to each other, although we may need the confluence property (Lemma 2.4.5) established below to prove that.

As an example, we can type the function $\lambda \mathtt{x}{:}\mathbb{B} \Rightarrow \mathtt{if\ x\ then\ false\ else\ } \lambda \mathtt{y}{:}\mathbb{B} \Rightarrow \mathtt{y}$ with $\Pi \mathtt{x}{:}\mathbb{B}\mathtt{,if\ x\ then\ }\mathbb{B}\mathtt{\ else\ }\mathbb{B} \to \mathbb{B}$ using the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{x} : \mathbb{B} \vdash \mathtt{false} : \mathbb{B}}\text{T-Lit} \quad \mathbb{B} \equiv \mathtt{if\ true\ then\ }\mathbb{B}\mathtt{\ else\ }\mathbb{B} \to \mathbb{B} \quad \cdots}{\mathtt{x} : \mathbb{B} \vdash \mathtt{false} : \mathtt{if\ true\ then\ }\mathbb{B}\mathtt{\ else\ }\mathbb{B} \to \mathbb{B}}\text{T-Conv} \quad \cdots}{\mathtt{x} : \mathbb{B} \vdash \mathtt{if\ x\ then\ false\ else\ } \lambda \mathtt{y}{:}\mathbb{B} \Rightarrow \mathtt{y} : \mathtt{if\ x\ then\ }\mathbb{B}\mathtt{\ else\ }\mathbb{B} \to \mathbb{B}}\text{T-If} \quad \dfrac{}{\cdot \vdash \mathbb{B} :: *}\text{K-Bool}}{\cdot \vdash \lambda \mathtt{x}{:}\mathbb{B} \Rightarrow \mathtt{if\ x\ then\ false\ else\ } \lambda \mathtt{y}{:}\mathbb{B} \Rightarrow \mathtt{y} : \Pi \mathtt{x}{:}\mathbb{B}\mathtt{,if\ x\ then\ }\mathbb{B}\mathtt{\ else\ }\mathbb{B} \to \mathbb{B}}\text{T-Abs}$$

Only the derivation of the `then` branch in the T-If application is shown; the `else` branch is similar, and the condition is easy to type. We choose the motive $\mathtt{if\ z\ then\ }\mathbb{B}\mathtt{\ else\ }\mathbb{B} \to \mathbb{B}$ when applying T-If. After instantiating $\mathtt{z}$ with `true` in the type of the `then` branch, this type is converted to $\mathbb{B}$ by applying T-Conv.

The expressiveness and strong guarantees provided by dependently typed languages come at a price, one of which being a much more intricate metatheory. In the rest of this section, we will prove the standard preservation property as an example of these complexities. As a

side note, all typing contexts (Γ) in this dissertation (as well as in the Coq mechanization) are modeled as maps with extensional equality. As a result, in contrast to modeling typing contexts as ordered lists, as is done in some works, our statements and proofs have a simpler treatment to typing contexts.

First, we must establish that parallel reduction is *confluent*, also known as the *Church-Rosser property*. To this end, we show substitution preserves parallel reduction.

Lemma 2.4.1. *If* $\mathtt{s} \Rrightarrow \mathtt{s}'$, *then* $\mathtt{[s/x]e} \Rrightarrow \mathtt{[s'/x]e}$.

Proof. By routine induction on the structure of $\mathtt{e}$. □

Lemma 2.4.2 (Substitution preserves parallel reduction)**.** *If* $\mathtt{e} \Rrightarrow \mathtt{e}'$ *and* $\mathtt{s} \Rrightarrow \mathtt{s}'$, *then* $\mathtt{[s/x]e} \Rrightarrow \mathtt{[s'/x]e'}$.

Proof. By routine induction on the derivation of $\mathtt{e} \Rrightarrow \mathtt{e}'$. □

Lemma 2.4.2 can be extended to a lemma about type equivalence.

Lemma 2.4.3 (Substitution preserves type equivalence)**.** *If* $\mathtt{e} \equiv \mathtt{e}'$ *and* $\mathtt{s} \equiv \mathtt{s}'$, *then* $\mathtt{[s/x]e} \equiv \mathtt{[s'/x]e'}$.

Proof. Easily obtained from Lemma 2.4.2. □

We then prove an important diamond property [47] that says the reductions of a term can converge in one reduction step. The proof is fairly straightforward (albeit tedious).

Lemma 2.4.4 (Diamond)**.** *If* $\mathtt{e} \Rrightarrow \mathtt{e}_1$ *and* $\mathtt{e} \Rrightarrow \mathtt{e}_2$, *then* $\mathtt{e}_1 \Rrightarrow \mathtt{e}'$ *and* $\mathtt{e}_2 \Rrightarrow \mathtt{e}'$ *for some* $\mathtt{e}'$.

Proof. We proceed by induction on the first derivation and then inverting the second one.

Case R-AppCgr: Suppose $\mathtt{e}_1\ \mathtt{e}_2 \Rrightarrow \mathtt{e}'_1\ \mathtt{e}'_2$ because $\mathtt{e}_1 \Rrightarrow \mathtt{e}'_1$ and $\mathtt{e}_2 \Rrightarrow \mathtt{e}'_2$. Inverting the second derivation gives us three possibilities.

First, $\mathtt{e}_1\ \mathtt{e}_2 \Rrightarrow \mathtt{e}_1\ \mathtt{e}_2$ because of R-Refl. This case is trivial.

Second, $\mathtt{e}_1\ \mathtt{e}_2 \Rrightarrow \mathtt{u}_1\ \mathtt{u}_2$ for some $\mathtt{u}_1$ and $\mathtt{u}_2$ because of R-AppCgr and $\mathtt{e}_1 \Rrightarrow \mathtt{u}_1$ and $\mathtt{e}_2 \Rrightarrow \mathtt{u}_2$. In this case, by the induction hypotheses, $\mathtt{u}_1 \Rrightarrow \mathtt{t}_1$ and $\mathtt{e}'_1 \Rrightarrow \mathtt{t}_1$ for some $\mathtt{t}_1$. Similarly, $\mathtt{u}_2 \Rrightarrow \mathtt{t}_2$ and $\mathtt{e}'_2 \Rrightarrow \mathtt{t}_2$ for some $\mathtt{t}_2$. Therefore, $\mathtt{e}'_1\ \mathtt{e}'_2$ and $\mathtt{u}_1\ \mathtt{u}_2$ reduce to the same expression $\mathtt{t}_1\ \mathtt{t}_2$ by applying R-AppCgr.

Finally, $\mathtt{e}_1 = \lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{u}_1$ and $(\lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{u}_1)\ \mathtt{e}_2 \Rrightarrow [\mathtt{u}_1'/\mathtt{x}]\mathtt{u}_2'$ for some $\mathtt{u}_1$, $\mathtt{u}_1'$ and $\mathtt{u}_2'$ because of R-App and $\mathtt{u}_1 \Rrightarrow \mathtt{u}_1'$ and $\mathtt{e}_2 \Rrightarrow \mathtt{u}_2'$. By assumption, $\lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{u}_1 \Rrightarrow \mathtt{e}_1'$, so $\mathtt{e}_1' = \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{s}$ for some τ_1 and $\mathtt{s}$ such that $\tau \Rrightarrow \tau_1$ and $\mathtt{u}_1 \Rrightarrow \mathtt{s}$ because of R-AbsCgr or R-Refl. Since $\lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{u}_1 \Rrightarrow \lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{u}_1'$ from R-AbsCgr, by the induction hypothesis, $\lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{u}_1' \Rrightarrow \lambda \mathtt{x}{:}\tau' \Rightarrow \mathtt{t}_1$ and $\lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{s} \Rrightarrow \lambda \mathtt{x}{:}\tau' \Rightarrow \mathtt{t}_1$ for some τ' and $\mathtt{t}_1$. We also have $\mathtt{u}_1' \Rrightarrow \mathtt{t}_1$ and $\mathtt{s} \Rrightarrow \mathtt{t}_1$ by inverting these reductions. On the other hand, $\mathtt{e}_2' \Rrightarrow \mathtt{t}_2$ and $\mathtt{u}_2' \Rrightarrow \mathtt{t}_2$ for some $\mathtt{t}_2$ by the induction hypothesis. It then follows that $\mathtt{e}_1'\ \mathtt{e}_2'$, i.e., $(\lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{s})\ \mathtt{e}_2'$ reduces to $[\mathtt{t}_1/\mathtt{x}]\mathtt{t}_2$ by R-App, and $[\mathtt{u}_1'/\mathtt{x}]\mathtt{u}_2'$ also reduces to $[\mathtt{t}_1/\mathtt{x}]\mathtt{t}_2$ by Lemma 2.4.2, completing the proof of R-AppCgr case.

The proofs of other cases are similar and left as an exercise for the reader. □

From the diamond property, we can easily obtain confluence of parallel reduction.

Lemma 2.4.5 (Confluence)**.** *If* $\mathtt{e} \Rrightarrow^* \mathtt{e}_1$ *and* $\mathtt{e} \Rrightarrow^* \mathtt{e}_2$*, then* $\mathtt{e}_1 \Rrightarrow^* \mathtt{e}'$ *and* $\mathtt{e}_2 \Rrightarrow^* \mathtt{e}'$ *for some* $\mathtt{e}'$*.*

Proof. Proceed by induction on the reflexive and transitive closure of parallel reduction and Lemma 2.4.4. □

We can finally show that the type equivalence defined previously is indeed an equivalence relation.

Lemma 2.4.6 (Type equivalence)**.** *The definition of type equivalence* $\equiv$ *is an equivalence relation.*

Proof. The confluence property, i.e., Lemma 2.4.5, is necessary for proving transitivity of $\equiv$. Reflexivity and symmetry are trivial. □

Some standard lemmas are needed to prove preservation.

Lemma 2.4.7 (Weakening)**.** *If* $\Gamma \vdash \mathtt{e} : \tau$ *and* $\Gamma \subseteq \Gamma'$*, then* $\Gamma' \vdash \mathtt{e} : \tau$*.*

If $\Gamma \vdash \tau :: *$ *and* $\Gamma \subseteq \Gamma'$*, then* $\Gamma' \vdash \tau :: *$*.*

Proof. We prove these two statements simultaneously by induction on the typing and kinding derivations. □

Lemma 2.4.8 (Substitution). *If* $\mathtt{x} : \tau', \Gamma \vdash \mathtt{e} : \tau$ *and* $\Gamma \vdash \mathtt{s} : \tau'$*, then* $\Gamma \vdash \mathtt{[s/x]e} : \mathtt{[s/x]}\tau$.

If $\mathtt{x} : \tau', \Gamma \vdash \tau :: *$ *and* $\Gamma \vdash \mathtt{s} : \tau'$*, then* $\Gamma \vdash \mathtt{[s/x]}\tau :: *$.

Proof. By routine (mutual) induction on the typing and kinding derivations. The proof requires the weakening lemma, i.e., Lemma 2.4.7. □

Lemma 2.4.9 (Regularity). *If* $\Gamma \vdash \mathtt{e} : \tau$*, then* $\Gamma \vdash \tau :: *$.

Proof. By induction on the typing derivation. Most cases are trivial, but the case of T-App depends on the kinding part of Lemma 2.4.8. □

Lemma 2.4.10 (Type conversion in context). *If* $\mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e} : \tau$ *and* $\tau_1 \equiv \tau_2$ *with* $\Gamma \vdash \tau_2 :: *$*, then* $\mathtt{x} : \tau_2, \Gamma \vdash \mathtt{e} : \tau$.

If $\mathtt{x} : \tau_1, \Gamma \vdash \tau :: *$ *and* $\tau_1 \equiv \tau_2$ *with* $\Gamma \vdash \tau_2 :: *$*, then* $\mathtt{x} : \tau_2, \Gamma \vdash \tau :: *$.

Proof. By routine (mutual) induction on the typing and kinding derivations. Lemma 2.4.7, Lemma 2.4.8 and Lemma 2.4.9 are needed. □

Lemma 2.4.11 (Inversion). *If* $\Gamma \vdash \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e} : \Pi \mathtt{x}{:}\tau_1', \tau_2$*, then* $\tau_1 \equiv \tau_1'$ *and* $\mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e} : \tau_2$ *and* $\Gamma \vdash \tau_1 :: *$.

Proof. By assumption, $\Gamma \vdash \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e} : \tau$ for some τ such that $\tau \equiv \Pi \mathtt{x}{:}\tau_1', \tau_2$. We proceed by induction on this new typing derivation. Most cases are vacuous except for T-Abs and T-Conv. The case of T-Conv is trivial by the induction hypothesis, and it relies on transitivity of type equivalence (Lemma 2.4.6). In the case of T-Abs, $\Gamma \vdash \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e} : \Pi \mathtt{x}{:}\tau_1, \tau_2'$, so $\tau_1 \equiv \tau_1'$ and $\tau_2 \equiv \tau_2'$. It suffices to prove $\mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e} : \tau_2$, which follows from T-Conv and the induction hypothesis. □

Finally, we prove that parallel reduction preserves types.

Theorem 2.4.12 (Preservation). *If* $\Gamma \vdash \mathtt{e} : \tau$ *and* $\mathtt{e} \Rrightarrow \mathtt{e}'$*, then* $\Gamma \vdash \mathtt{e}' : \tau$.

If $\Gamma \vdash \tau :: *$ *and* $\tau \Rrightarrow \tau'$*, then* $\Gamma \vdash \tau' :: *$.

Proof. We prove these two statements simultaneously by (mutual) induction on the typing and kinding derivations. The cases of T-Lit, T-Conv and K-Bool are trivial. We omit

proofs of the side conditions about well-kindedness (e.g., when applying T-Conv), which are easily discharged by the kinding rules or by Lemma 2.4.9.

Case T-Abs: By assumption, $\lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e} \Rrightarrow \lambda \mathtt{x}{:}\tau_1' \Rightarrow \mathtt{e}'$ for some τ_1' and $\mathtt{e}'$ due to R-Refl or R-AbsCgr, as well as $\tau_1 \Rrightarrow \tau_2$ and $\mathtt{e} \Rrightarrow \mathtt{e}'$. By the induction hypothesis, $\mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e}' : \tau_2$, from which Lemma 2.4.10 gives $\mathtt{x} : \tau_1', \Gamma \vdash \mathtt{e}' : \tau_2$ because $\tau_1 \equiv \tau_1'$ and $\Gamma \vdash \tau_1' :: *$ again by the induction hypothesis. We then have $\Gamma \vdash \lambda \mathtt{x}{:}\tau_1' \Rightarrow \mathtt{e}' : \Pi \mathtt{x}{:}\tau_1', \tau_2$ by T-Abs. Finally, T-Conv gives us $\Gamma \vdash \lambda \mathtt{x}{:}\tau_1' \Rightarrow \mathtt{e}' : \Pi \mathtt{x}{:}\tau_1, \tau_2$ as required, because $\Pi \mathtt{x}{:}\tau_1', \tau_2 \equiv \Pi \mathtt{x}{:}\tau_1, \tau_2$.

Case T-App: We consider two possible derivations of parallel reduction.

First, $\mathtt{e}_2\ \mathtt{e}_1 \Rrightarrow \mathtt{e}_2'\ \mathtt{e}_1'$ with $\mathtt{e}_2 \Rrightarrow \mathtt{e}_2'$ and $\mathtt{e}_1 \Rrightarrow \mathtt{e}_1'$ for some $\mathtt{e}_2'$ and $\mathtt{e}_1'$, due to R-Refl or R-AppCgr. In this case, $\Gamma \vdash \mathtt{e}_2'\ \mathtt{e}_1' : [\mathtt{e}_1'/\mathtt{x}]\tau_2$ by T-App and induction hypotheses. Since $[\mathtt{e}_1'/\mathtt{x}]\tau_2 \equiv [\mathtt{e}_1/\mathtt{x}]\tau_2$ by Lemma 2.4.3, we have $\Gamma \vdash \mathtt{e}_2'\ \mathtt{e}_1' : [\mathtt{e}_1/\mathtt{x}]\tau_2$ from T-Conv as desired.

Second, $\mathtt{e}_2 = \lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{s}$, and $(\lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{s})\ \mathtt{e}_1 \Rrightarrow [\mathtt{e}_1'/\mathtt{x}]\mathtt{s}'$ for some τ, $\mathtt{e}_1'$, $\mathtt{s}$ and $\mathtt{s}'$, with $\mathtt{s} \Rrightarrow \mathtt{s}'$ and $\mathtt{e}_1 \Rrightarrow \mathtt{e}_1'$, due to R-App. It follows that $\lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{s} \Rrightarrow \lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{s}'$, from which $\Gamma \vdash \lambda \mathtt{x}{:}\tau \Rightarrow \mathtt{s}' : \Pi \mathtt{x}{:}\tau_1, \tau_2$ by the induction hypothesis. We then have $\mathtt{x} : \tau, \Gamma \vdash \mathtt{s}' : \tau_2$ with $\tau \equiv \tau_1$ and $\Gamma \vdash \tau :: *$ by Lemma 2.4.11 (inversion lemma). It then follows that $\Gamma \vdash [\mathtt{e}'/\mathtt{x}]\mathtt{s}' : [\mathtt{e}'/\mathtt{x}]\tau_2$ by Lemma 2.4.8, because $\Gamma \vdash \mathtt{e}_1' : \tau$ by T-Conv and the induction hypothesis. Finally, T-Conv gives us $\Gamma \vdash [\mathtt{e}_1'/\mathtt{x}]\mathtt{s}' : [\mathtt{e}_1/\mathtt{x}]\tau_2$ since $[\mathtt{e}_1'/\mathtt{x}]\tau_2 \equiv [\mathtt{e}_1/\mathtt{x}]\tau_2$ by Lemma 2.4.3.

Case T-If: Again, we consider two possible derivations of parallel reduction.

First, $\mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \Rrightarrow \mathtt{if}\ \mathtt{e}_0'\ \mathtt{then}\ \mathtt{e}_1'\ \mathtt{else}\ \mathtt{e}_2'$ for some $\mathtt{e}_0'$, $\mathtt{e}_1'$ and $\mathtt{e}_2'$, with $\mathtt{e}_0 \Rrightarrow \mathtt{e}_0'$, $\mathtt{e}_1 \Rrightarrow \mathtt{e}_1'$ and $\mathtt{e}_2 \Rrightarrow \mathtt{e}_2'$, because of R-Refl or R-IfCgr. By the induction hypotheses and T-If, we obtain $\Gamma \vdash \mathtt{if}\ \mathtt{e}_0'\ \mathtt{then}\ \mathtt{e}_1'\ \mathtt{else}\ \mathtt{e}_2' : [\mathtt{e}_0'/\mathtt{z}]\tau$. The kinding side condition of T-If can be discharged by Lemma 2.4.1 and the induction hypothesis. T-Conv then gives us $\Gamma \vdash \mathtt{if}\ \mathtt{e}_0'\ \mathtt{then}\ \mathtt{e}_1'\ \mathtt{else}\ \mathtt{e}_2' : [\mathtt{e}_0/\mathtt{z}]\tau$ because $[\mathtt{e}_1/\mathtt{z}]\tau \equiv [\mathtt{e}_0/\mathtt{z}]\tau$ by Lemma 2.4.3.

Second, $\mathtt{e}_0 = \mathtt{true}$ and $\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ \mathtt{e}_0\ \mathtt{else}\ \mathtt{e}_1 \Rrightarrow \mathtt{e}_1'$ for some $\mathtt{e}_1'$ with $\mathtt{e}_1 \Rrightarrow \mathtt{e}_1'$, due to R-IfTrue (the case of $\mathtt{e}_0 = \mathtt{false}$ due to R-IfFalse is similar). In this case, we want to show $\Gamma \vdash \mathtt{e}_1' : [\mathtt{true}/\mathtt{z}]\tau$, but that is immediate from the induction hypothesis.

The remaining cases about kinding derivations are similar: the proof for K-Pi is similar to the case of T-Abs, and the proof for K-If is similar to T-If (but easier as substitutions do not occur in kinds). □

Since parallel reduction is a more liberal relation than the call-by-value or call-by-name semantics, these standard semantics also inherit the preservation property.

3. OBLIVIOUS ALGEBRAIC DATA TYPES

As discussed in Chapter 1, to support private data structures, we have to ensure no private information, including data's structural information if policy makers choose to hide it, can be inferred by observing the representation or manipulation of private data. In this chapter, 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.

In summary, this chapter 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 $\lambda_{\texttt{OADT}}$, a core calculus for writing oblivious programs using OADTs, whose strong type system ensures computations are secure.

The core calculus $\lambda_{\texttt{OADT}}$ and its metatheory have been mechanically formalized in the Coq proof assistant. An artifact containing both these developments is publicly available [48].

3.1 Overview

To illustrate our approach, consider the simple function in Figure 3.1, 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

```
data tree = Leaf | Node ℤ tree tree

fn lookup (x : ℤ) (t : tree) : 𝔹 =
  match t with
  | Leaf ⇒ false
  | Node y tl tr ⇒
    if x ≤ y then if y ≤ x then true
                  else lookup x tl
    else lookup x tr
```

Figure 3.1. Lookup element in a search tree

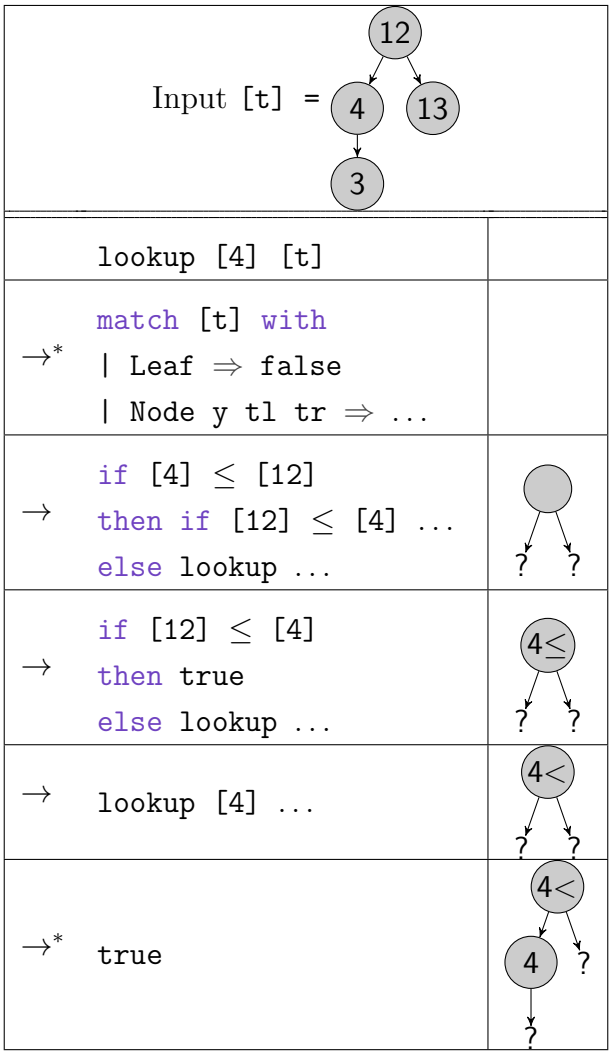


Figure 3.2. Execution trace of `lookup [4] [t]`. The columns show the current state of the program, and information learned by the owner of the lookup key, respectively.

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 (Section 2.1). 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.

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 3.2, 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 `match` statement that checks whether to recurse, and the `if` statement that decides which subtree to recurse on. As the fourth row illustrates, the branch `match` 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.

3.1.1 Encoding Private Data and Policies

The first component to our solution is our representation of both private data and the public information about the data that can be freely shared. We call this publicly shared information a *public view*, reflecting that it is some projection of the full data. Each public view corresponds to a *privacy policy* governing the data. Formally, policies are encoded

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

```
obliv tree̲ (k : ℕ) =
  if k = 0
  then 𝟙
  else 𝟙 +̂ ℤ̂ × tree̲ (k−1) × tree̲ (k−1)
```

(a) Maximum depth as public view

```
obliv tree̲′ (s : spine) =
  match s with
  | SLeaf ⇒ 𝟙
  | SNode sl sr ⇒
    𝟙 +̂ ℤ̂ × tree̲′ sl × tree̲′ sr
```

(b) Upper bound of spine as public view

Figure 3.3. Oblivious trees

as *oblivious algebraic data types* (OADTs), dependent types that take a public view as a parameter. The body of an OADT is the type of the private components of a data type, which are built using *oblivious* (i.e., secure) type formers, e.g., oblivious fixed-width integer ($\hat{\mathbb{Z}}$) and oblivious sum ($\hat{+}$). By convention, we use $\hat{\cdot}$ to denote the oblivious version of something. Essentially, an OADT is a type-level function that maps the public view of a value to its *private representation*, i.e., the shape of its private component. Section 3.2.4 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 3.2, the oblivious integer `[4]` is *indistinguishable* from `[12]`, for example.

Figure 3.3a gives an example of an oblivious tree whose public view is its maximum depth. In general, a public view can be any public data type. We say `tree` is the *public type* or public counterpart of the OADT $\widehat{\texttt{tree}}$. The key idea behind oblivious ADTs is to construct a representation of private data from the public view. As a consequence, private values with the same public view are *indistinguishable* to an attacker, as their private representation is completely determined by the public view. For example, all oblivious trees with a maximum depth of two have the same private representation, regardless of the actual depth of the tree:

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

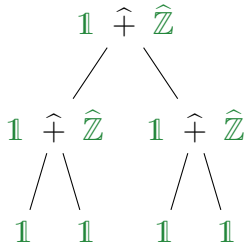


Figure 3.4. Oblivious tree with a maximum depth of two

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 3.4. 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 $\widehat{+}$, as the left injection (e.g., a `Leaf`) and the right injection (e.g., a `Node`) 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: an empty tree, singleton tree, a tree with two elements, or a complete tree of depth 2 all appear the same to an attacker.

Figure 3.3b 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 3.3a, but it also enjoys a more efficient representation, as it requires less padding than a complete tree.

Figure 3.5 presents an oblivious type for a simplified version of the medical record with the "either-or" policy from the introduction (Chapter 1). A `patient` record consists of their ID, age, height and weight. The public view in this example consists of either a patient's ID (`Known_id`), or their height and weight (`Known_data`); their age is always private. The corresponding oblivious type $\widehat{\texttt{patient}}$ is straightforward: it is the oblivious data that has been omitted from the public view. If the ID is disclosed, for example, then the oblivious type is essentially an encrypted version of the remaining 3 fields. Oblivious ADTs are expressive enough to directly support this kind of "either-or" policy.

```
// Id, age, height and weight
data patient = Patient ℤ ℤ ℤ ℤ
data patient_view = Known_id ℤ
                  | Known_data ℤ ℤ
obliv patient (v : patient_view) =
  match v with
  | Known_id _ ⇒ ℤ̂ ×̂ ℤ̂ ×̂ ℤ̂
  | Known_data _ _ ⇒ ℤ̂ ×̂ ℤ̂
```

Figure 3.5. Either-or policy as an OADT

Conceptually, OADTs generalize the notion of secure fixed-width integers to secure structured data, as illustrated in Figure 3.6. Every fixed-width integer (of type $\mathbb{Z}$) can be sent to its secure value in $\widehat{\mathbb{Z}}$ by "encryption", and a secure integer can be converted back to $\mathbb{Z}$ by "decryption". We call these conversion functions *section* (e.g., $\widehat{\mathbb{Z}}$`#s`) and *retraction* (e.g., $\widehat{\mathbb{Z}}$`#r`). The names reflect their expected semantics: applying retraction to the section of a value should produce the same value.

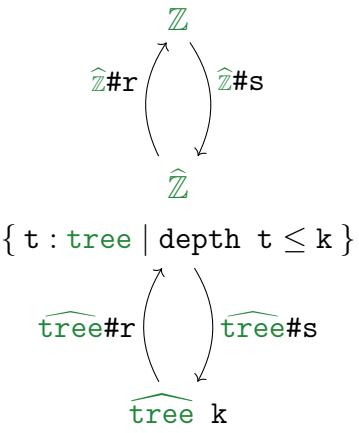


Figure 3.6. Public and oblivious types

Importantly, while the oblivious integer type $\widehat{\mathbb{Z}}$ does not appear to have much structure, it nonetheless has an implicit policy: the public view of an integer is its bit width. If we use 32-bit integers, for example, $\mathbb{Z}$ is the set of all integers whose bit width is 32, and $\widehat{\mathbb{Z}}$ is the set of their "encrypted" values, related by a pair of conversion functions. Similarly, $\widehat{\texttt{tree}}$ `k` consists of the secure encodings of trees that have at most `k` layers. Like $\widehat{\mathbb{Z}}$, $\widehat{\texttt{tree}}$ is equipped with a section function, $\widehat{\texttt{tree}}$`#s`, and a retraction function, $\widehat{\texttt{tree}}$`#r`, which convert public values of `tree` to their oblivious counterparts and back. Crucially, just as the oblivious integers in $\widehat{\mathbb{Z}}$ are indistinguishable, the elements of $\widehat{\texttt{tree}}$ `k` are also indistinguishable.

3.1.2 Enforcing Policies

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 `if [4]` $\leq$ `[12] then ... else ...`, 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 $\lambda_{\texttt{OADT}}$, a pure functional language for writing secure computations over OADTs. $\lambda_{\texttt{OADT}}$ 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 $\lambda_{\texttt{OADT}}$ expression, which compares two secure integers to determine what value to return:

$$\texttt{mux}\ (\texttt{[0]}\ \widehat{\leq}\ \texttt{[1]})\ (\texttt{[2]}\ \widehat{+}\ \texttt{[3]})\ (\texttt{[4]}\ \widehat{+}\ \texttt{[5]})$$

We use notations $\widehat{+}$ and $\widehat{\leq}$ for the oblivious versions of $+$ and $\leq$, such that $\texttt{[4]}\ \widehat{+}\ \texttt{[3]} \longrightarrow \texttt{[7]}$ and $\texttt{[4]}\ \widehat{\leq}\ \texttt{[3]} \longrightarrow \texttt{[false]}$.[2] Here, mux (short for *multiplexer*) 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 private condition. To do so, it fully evaluates *both* branches before stepping to the final (oblivious) result:

$$\begin{aligned}
&\texttt{mux}\ (\texttt{[0]}\ \widehat{\leq}\ \texttt{[1]})\ (\texttt{[2]}\ \widehat{+}\ \texttt{[3]})\ (\texttt{[4]}\ \widehat{+}\ \texttt{[5]}) \\
\longrightarrow\ &\texttt{mux}\ \texttt{[true]}\ (\texttt{[2]}\ \widehat{+}\ \texttt{[3]})\ (\texttt{[4]}\ \widehat{+}\ \texttt{[5]}) \\
\longrightarrow\ &\texttt{mux}\ \texttt{[true]}\ \texttt{[5]}\ (\texttt{[4]}\ \widehat{+}\ \texttt{[5]}) \longrightarrow \texttt{mux}\ \texttt{[true]}\ \texttt{[5]}\ \texttt{[9]} \longrightarrow \texttt{[5]}
\end{aligned}$$

Replacing [0] with [6] 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:

$$\begin{aligned}
&\texttt{mux}\ (\texttt{[6]}\ \widehat{\leq}\ \texttt{[1]})\ (\texttt{[2]}\ \widehat{+}\ \texttt{[3]})\ (\texttt{[4]}\ \widehat{+}\ \texttt{[5]}) \\
\longrightarrow\ &\texttt{mux}\ \texttt{[false]}\ (\texttt{[2]}\ \widehat{+}\ \texttt{[3]})\ (\texttt{[4]}\ \widehat{+}\ \texttt{[5]}) \\
\longrightarrow\ &\texttt{mux}\ \texttt{[false]}\ \texttt{[5]}\ (\texttt{[4]}\ \widehat{+}\ \texttt{[5]}) \longrightarrow \texttt{mux}\ \texttt{[false]}\ \texttt{[5]}\ \texttt{[9]} \longrightarrow \texttt{[9]}
\end{aligned}$$

The oblivious sum pattern matching statement ($\widehat{\texttt{match}}$) behaves similarly, with the additional wrinkle that the pattern variables of the "wrong" branch are bound to some arbitrary oblivious values, which Section 3.2 explains in full detail.

$\lambda_{\texttt{OADT}}$ is equipped with a security-type system [27], to ensure the correct use of its secure operations. The full details of this type system can be found in Section 3.2.3, 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 $\mathbb{B}\ \widehat{+}\ \mathbb{Z}$.

[2]↑We abuse the notation $\widehat{+}$ to mean both oblivious sum and oblivious integer addition.

```
fn lookup0 (x : ℤ̂) (k : ℕ) : tree̲ k → 𝔹̂ =
  if k = 0
  then λ_ ⇒ 𝔹̂#s false
  else λt ⇒ match̲ t with
            | inl̲ _ ⇒ 𝔹̂#s false
            | inr̲ (y, tl, tr) ⇒
              mux (x ≤̂ y)
                  (mux (y ≤̂ x) (𝔹̂#s true) (lookup0 x (k−1) tl))
                  (lookup0 x (k−1) tr)
```

Figure 3.7. Oblivious lookup function in $\lambda_{\texttt{OADT}}$

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 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]` $\hat{\mathbb{B}}$ $\hat{\mathbb{Z}}$ is not a valid type in $\lambda_{\texttt{OADT}}$.

Figure 3.7 presents an oblivious implementation of the `lookup` function for the oblivious tree from Figure 3.3a. 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 $\hat{\mathbb{B}}$`#s` operation (i.e., boolean section), which acts like a coercion from public booleans to oblivious booleans. This implementation is guaranteed to be secure, although it is not quite pleasing to write due to the intermixing of privacy policies and program logic. Chapter 4 will introduce a more ergonomic language that allows programmers to write functionality just like the "standard" implementation of `lookup` in Figure 3.1.

3.1.3 Performance Implication of 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, where attackers can observe every program state in executions.

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 $\lambda_{\texttt{OADT}}$. 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, organizational hierarchies, probability tree diagram, query languages, and decision trees. Complex policies, such as the either-or policy, can be encoded as OADTs as well. Using the language introduced in Chapter 4, users can quickly prototype secure computation over structured data, and explore the impact of different public views on a computation.

3.2 $\lambda_{\texttt{OADT}}$, Formally

This section formalizes $\lambda_{\texttt{OADT}}$, a core calculus for programming with OADTs. The calculus described in this section has been mechanized in the Coq proof assistant and can be found in the publicly available artifact [48].

e,τ	::=		Expressions:
	|	$\mathbb{1}$ | $\mathbb{B}$ | $\widehat{\mathbb{B}}$ | τ×τ | τ+τ | τ$\widehat{+}$τ	simple types
	|	Πx:τ,τ	dependent function type
	|	x	variable
	|	() | true | false	unit and boolean values
	|	λx:τ⇒e	function abstraction
	|	let x = e in e	let binding
	|	e e | $\widehat{\text{T}}$ 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
	|	ι_b<τ> e | $\widehat{\iota_b}$<τ> e	(oblivious) sum injection
	|	match e with x⇒e|x⇒e	sum elimination
	|	$\widehat{\text{match}}$ e with x⇒e|x⇒e	oblivious sum elimination
	|	fold<T> e | unfold<T> e	iso-recursive type intro. and elim.
	|	$\widehat{\mathbb{B}}$#s e	boolean section
	|	[b] | [ι_b<$\widehat{\omega}$> $\hat{\text{v}}$]	runtime boxed values
D	::=		Global Definitions:
	|	data T = τ	algebraic data type definition
	|	fn x:τ = e	(recursive) function definition
	|	obliv $\widehat{\text{T}}$ (x:τ) = τ	(recursive) oblivious type definition
$\widehat{\omega}$	::=	$\mathbb{1}$ | $\widehat{\mathbb{B}}$ | $\widehat{\omega}$×$\widehat{\omega}$ | $\widehat{\omega}\widehat{+}\widehat{\omega}$	Oblivious Type Values
$\hat{\text{v}}$	::=	() | [b] | ($\hat{\text{v}}$,$\hat{\text{v}}$) | [ι_b<$\widehat{\omega}$> $\hat{\text{v}}$]	Oblivious Values
v	::=	$\hat{\text{v}}$ | b | (v,v) | λx:τ⇒e	Values
	|	ι_b<τ> v | fold<T> v	

Figure 3.8. λ_{OADT} syntax

3.2.1 Syntax

The core syntax of λ_{OADT} is shown in Figure 3.8. 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 4.3. 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 `fn`, respectively. Using a global set of function definitions naturally supports general recursion and mutual recursion. When possible, we use `x` for function names, `T` for public ADT names, and $\widehat{\texttt{T}}$ for the names of oblivious ADTs.

Types in $\lambda_{\texttt{OADT}}$ include dependent function types (Π), sums ($+$), products ($\times$), 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 $\lambda_{\texttt{OADT}}$, 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 [29]. $\lambda_{\texttt{OADT}}$ supports type-level computation via large elimination, allowing users to compute types from terms using application, `let`, `if`, and `match`. 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 $\lambda_{\texttt{OADT}}$ are largely standard. A subscript distinguishes between left or right injection (ι_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 $\iota_{\texttt{true}}$ ($\iota_{\texttt{false}}$) and $\pi_{\texttt{true}}$ ($\pi_{\texttt{false}}$). Injections are annotated with their full type, in order to completely determine its data representation. $\lambda_{\texttt{OADT}}$ 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). The atomic conditional `mux` is the core oblivious construct in $\lambda_{\texttt{OADT}}$; as Section 3.1 discussed, `mux` fully evaluates both of its branches before taking a single atomic step to the correct branch. Other oblivious constructs include boolean section $\widehat{\mathbb{B}}$`#s`, which builds an oblivious boolean from its argument, and constructors ($\widehat{\iota}_b$) and an eliminator ($\widehat{\texttt{match}}$) for oblivious sums.

In addition to the expected sorts of public values, $\lambda_{\texttt{OADT}}$ also includes oblivious values for booleans (`[b]`) and sums (`[`ι_b`<`$\widehat{\omega}$`> v̂]`). 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 $\widehat{\mathbb{B}}$`#s` or $\widehat{\iota}_b$. As Section 3.1 discussed, these "boxed" values represent secure data which cannot be observed by an adversary. Since $\lambda_{\texttt{OADT}}$ 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.

3.2.2 Semantics

Figure 3.9 defines a relation for the small-step operational semantics of $\lambda_{\texttt{OADT}}$. The judgment of this relation has the form $\Sigma \vdash \texttt{e} \longrightarrow \texttt{e}'$, and is read as "$\texttt{e}$ steps to $\texttt{e}'$ under the global context Σ". Since a $\lambda_{\texttt{OADT}}$ program is evaluated under a fixed global context, we often abbreviate this judgment as $\texttt{e} \longrightarrow \texttt{e}'$, referring to Σ only when needed. The S-Ctx rule uses the evaluation contexts ($\mathcal{E}$) defined at the bottom of Figure 3.9 to evaluate subexpressions. While these evaluation contexts are not inductively defined, it is possible to recursively apply S-Ctx when evaluating subterms.

Most of the non-oblivious reduction rules are standard. For brevity, several of the rules use the `ite` meta-function, which returns $\texttt{e}_1$ when its first argument is `true`, and $\texttt{e}_2$ otherwise. S-If is essentially the following two rules, for example:

S-IfTrue

$$\texttt{if true then } \texttt{e}_1 \texttt{ else } \texttt{e}_2 \longrightarrow \texttt{e}_1$$

S-IfFalse

$$\texttt{if false then } \texttt{e}_1 \texttt{ else } \texttt{e}_2 \longrightarrow \texttt{e}_2$$

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 $\widehat{\iota}$ to values before the S-OInj rule 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-OMatch, 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, $\widehat{\texttt{v}} \Leftarrow \widehat{\omega}$, which is also shown in

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

S-Ctx
$$\frac{\texttt{e} \longrightarrow \texttt{e}'}{\mathcal{E}[\texttt{e}] \longrightarrow \mathcal{E}[\texttt{e}']}$$

S-Fun
$$\frac{\texttt{fn x:}\tau\texttt{ = e} \in \Sigma}{\texttt{x} \longrightarrow \texttt{e}}$$

S-OADT
$$\frac{\texttt{obliv } \widehat{\texttt{T}}\texttt{ (x:}\tau\texttt{) = }\tau' \in \Sigma}{\widehat{\texttt{T}}\texttt{ v} \longrightarrow [\texttt{v/x}]\tau'}$$

S-App
$$\frac{}{(\lambda\texttt{x:}\tau\Rightarrow\texttt{e})\texttt{ v} \longrightarrow [\texttt{v/x}]\texttt{e}}$$

S-Let
$$\frac{}{\texttt{let x = v in e} \longrightarrow [\texttt{v/x}]\texttt{e}}$$

S-If
$$\frac{}{\texttt{if b then e}_1\texttt{ else e}_2 \longrightarrow \texttt{ite(b,e}_1\texttt{,e}_2\texttt{)}}$$

S-Match
$$\frac{}{\texttt{match }\iota_b\texttt{<}\tau\texttt{> v with x}\Rightarrow\texttt{e}_1\texttt{|x}\Rightarrow\texttt{e}_2 \longrightarrow \texttt{ite(b,}[\texttt{v/x}]\texttt{e}_1\texttt{,}[\texttt{v/x}]\texttt{e}_2\texttt{)}}$$

S-Proj
$$\frac{}{\pi_b\ (\texttt{v}_1\texttt{,v}_2) \longrightarrow \texttt{ite(b,v}_1\texttt{,v}_2\texttt{)}}$$

S-Unfold
$$\frac{}{\texttt{unfold<T> (fold<T}'\texttt{> v)} \longrightarrow \texttt{v}}$$

S-Sec
$$\frac{}{\widehat{\mathbb{B}}\texttt{\#s b} \longrightarrow \texttt{[b]}}$$

S-OInj
$$\frac{}{\widehat{\iota}_b\texttt{<}\widehat{\omega}\texttt{> }\hat{\texttt{v}} \longrightarrow [\iota_b\texttt{<}\widehat{\omega}\texttt{> }\hat{\texttt{v}}]}$$

S-Mux
$$\frac{}{\texttt{mux [b] v}_1\texttt{ v}_2 \longrightarrow \texttt{ite(b,v}_1\texttt{,v}_2\texttt{)}}$$

S-OMatch
$$\frac{\hat{\texttt{v}}_1 \Leftarrow \widehat{\omega}_1 \qquad \hat{\texttt{v}}_2 \Leftarrow \widehat{\omega}_2}{\widehat{\texttt{match}}\ [\iota_b\texttt{<}\widehat{\omega}_1\widehat{+}\widehat{\omega}_2\texttt{> }\hat{\texttt{v}}]\texttt{ with x}\Rightarrow\texttt{e}_1\texttt{|x}\Rightarrow\texttt{e}_2 \longrightarrow \begin{array}{l}\texttt{mux [b] ite(b,}[\hat{\texttt{v}}\texttt{/x}]\texttt{e}_1\texttt{,}[\hat{\texttt{v}}_1\texttt{/x}]\texttt{e}_1\texttt{)}\\ \qquad\qquad\texttt{ite(b,}[\hat{\texttt{v}}_2\texttt{/x}]\texttt{e}_2\texttt{,}[\hat{\texttt{v}}\texttt{/x}]\texttt{e}_2\texttt{)}\end{array}}$$

Evaluation Contexts

$\mathcal{E}$ ::=
- $\square\times\tau \mid \widehat{\omega}\times\square \mid \square\widehat{+}\tau \mid \widehat{\omega}\widehat{+}\square$
- $\texttt{let x = }\square\texttt{ in e} \mid \texttt{e }\square \mid \square\texttt{ v} \mid \widehat{\texttt{T}}\ \square$
- $(\square\texttt{,e}) \mid (\texttt{v,}\square) \mid \pi_b\ \square$
- $\iota_b\texttt{<}\tau\texttt{> }\square \mid \widehat{\iota}_b\texttt{<}\square\texttt{> e} \mid \widehat{\iota}_b\texttt{<}\widehat{\omega}\texttt{> }\square$
- $\texttt{fold<T> }\square \mid \texttt{unfold<T> }\square$
- $\texttt{if }\square\texttt{ then e else e}$
- $\texttt{match }\square\texttt{ with x}\Rightarrow\texttt{e|x}\Rightarrow\texttt{e}$
- $\widehat{\texttt{match}}\ \square\texttt{ with x}\Rightarrow\texttt{e|x}\Rightarrow\texttt{e}$
- $\texttt{mux }\square\texttt{ e e} \mid \texttt{mux v }\square\texttt{ e} \mid \texttt{mux v v }\square$
- $\widehat{\mathbb{B}}\texttt{\#s }\square$

$\boxed{\hat{\texttt{v}} \Leftarrow \widehat{\omega}}$

OT-Unit
$$\frac{}{() \Leftarrow \mathbb{1}}$$

OT-OBool
$$\frac{}{\texttt{[b]} \Leftarrow \widehat{\mathbb{B}}}$$

OT-Prod
$$\frac{\hat{\texttt{v}}_1 \Leftarrow \widehat{\omega}_1 \qquad \hat{\texttt{v}}_2 \Leftarrow \widehat{\omega}_2}{(\hat{\texttt{v}}_1\texttt{,}\hat{\texttt{v}}_2) \Leftarrow \widehat{\omega}_1\times\widehat{\omega}_2}$$

OT-OSum
$$\frac{\hat{\texttt{v}} \Leftarrow \texttt{ite(b,}\widehat{\omega}_1\texttt{,}\widehat{\omega}_2\texttt{)}}{[\iota_b\texttt{<}\widehat{\omega}_1\widehat{+}\widehat{\omega}_2\texttt{> }\hat{\texttt{v}}] \Leftarrow \widehat{\omega}_1\widehat{+}\widehat{\omega}_2}$$

Figure 3.9. $\lambda_{\texttt{OADT}}$ semantics

Figure 3.9. Equipped with this relation, S-OMatch can be straightforwardly reduced to a mux expression. To see how, consider the rule corresponding to the case where b is false:

$$\frac{\hat{\mathtt{v}}_1 \Leftarrow \widehat{\omega}_1 \qquad \hat{\mathtt{v}}_2 \Leftarrow \widehat{\omega}_2}{\widehat{\texttt{match}}\ \texttt{[inr<}\widehat{\omega}_1\texttt{+}\widehat{\omega}_2\texttt{>}\ \hat{\mathtt{v}}\texttt{]}\ \texttt{with}\ \mathtt{x}_1\Rightarrow\mathtt{e}_1\,|\,\mathtt{x}_2\Rightarrow\mathtt{e}_2 \longrightarrow \texttt{mux}\ \texttt{[false]}\ [\hat{\mathtt{v}}_1/\mathtt{x}_1]\mathtt{e}_1\ [\hat{\mathtt{v}}/\mathtt{x}_2]\mathtt{e}_2}$$

In the true branch of the resulting mux expression, the pattern variable $\mathtt{x}_1$ is instantiated with an arbitrary oblivious value, $\hat{\mathtt{v}}_1$, while the corresponding pattern variable in the false branch is instantiated with the actual payload $\hat{\mathtt{v}}$. Using this rule, the expression on the top below can step to either of the (indistinguishable) expressions on the bottom:

$$\widehat{\texttt{match}}\ \texttt{[inr<(}\widehat{\mathbb{B}}\times\widehat{\mathbb{B}}\texttt{)}\widehat{+}\widehat{\mathbb{B}}\texttt{>}\ \texttt{[false]]}\ \texttt{with}\ \mathtt{x}_1\Rightarrow\pi_2\ \mathtt{x}_1\,|\,\mathtt{x}_2\Rightarrow\mathtt{x}_2$$

$$\texttt{mux}\ \texttt{[false]}\ (\pi_2\ \texttt{([false],[true])})\ \texttt{[false]} \qquad \texttt{mux}\ \texttt{[false]}\ (\pi_2\ \texttt{([true],[false])})\ \texttt{[false]}$$

The pattern variables in the first branch can be substituted by any pair of oblivious booleans, e.g., ([false],[true]) or ([true],[false]).

3.2.3 Type System

The type system of $\lambda_{\texttt{OADT}}$ 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 $\lambda_{\texttt{OADT}}$ are augmented with a *security label* which constrains how information flows through a program:

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

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 3.10, 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 $\lambda_{\texttt{OADT}}$ public and oblivious labels are not compatible. We elide the security label of a kind when it is not relevant, e.g., $\Gamma \vdash \tau :: *$.

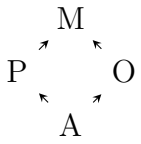


Figure 3.10. Semi-lattice on $\lambda_{\texttt{OADT}}$ kinds

Programs in $\lambda_{\texttt{OADT}}$ are typed using a pair of typing and kinding judgments; we denote these as $\Sigma;\Gamma \vdash \texttt{e} : \tau$ and $\Sigma;\Gamma \vdash \tau :: \kappa$, respectively. Figure 3.11 and Figure 3.12 give the kinding and typing rules for $\lambda_{\texttt{OADT}}$. 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 $\lambda_{\texttt{OADT}}$ are shown in Figure 3.11. 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, $\lambda_{\texttt{OADT}}$ 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 $*^{\texttt{P}}$ instead of $*^{\texttt{A}}$. Similarly, $\widehat{\mathbb{B}}+\widehat{\mathbb{B}}$ has to be kinded $*^{\texttt{M}}$, the join of $*^{\texttt{P}}$ and $*^{\texttt{O}}$, 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

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

$$\textsc{K-Unit}\ \frac{}{\Gamma \vdash \mathbb{1} :: *^{\mathsf{A}}} \qquad \textsc{K-Bool}\ \frac{}{\Gamma \vdash \mathbb{B} :: *^{\mathsf{P}}} \qquad \textsc{K-OBool}\ \frac{}{\Gamma \vdash \widehat{\mathbb{B}} :: *^{\mathsf{O}}} \qquad \textsc{K-ADT}\ \frac{\texttt{data T = } \tau \in \Sigma}{\Gamma \vdash \texttt{T} :: *^{\mathsf{P}}}$$

$$\textsc{K-Pi}\ \frac{\Gamma \vdash \tau_1 :: * \qquad \texttt{x} : \tau_1, \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \Pi \texttt{x:}\tau_1\texttt{,}\tau_2 :: *^{\mathsf{M}}} \qquad \textsc{K-Prod}\ \frac{\Gamma \vdash \tau_1 :: \kappa \qquad \Gamma \vdash \tau_2 :: \kappa}{\Gamma \vdash \tau_1 \times \tau_2 :: \kappa} \qquad \textsc{K-Sum}\ \frac{\Gamma \vdash \tau_1 :: \kappa \qquad \Gamma \vdash \tau_2 :: \kappa}{\Gamma \vdash \tau_1 + \tau_2 :: \kappa \sqcup *^{\mathsf{P}}}$$

$$\textsc{K-OSum}\ \frac{\Gamma \vdash \tau_1 :: *^{\mathsf{O}} \qquad \Gamma \vdash \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash \tau_1 \mathbin{\widehat{+}} \tau_2 :: *^{\mathsf{O}}} \qquad \textsc{K-OADT}\ \frac{\texttt{obliv } \widehat{\texttt{T}}\ \texttt{(x:}\tau\texttt{) = } \tau' \in \Sigma \qquad \Gamma \vdash \texttt{e} : \tau}{\Gamma \vdash \widehat{\texttt{T}}\ \texttt{e} :: *^{\mathsf{O}}}$$

$$\textsc{K-Let}\ \frac{\Gamma \vdash \texttt{e} : \tau \qquad \texttt{x} : \tau, \Gamma \vdash \tau' :: *^{\mathsf{O}}}{\Gamma \vdash \texttt{let x = e in } \tau' :: *^{\mathsf{O}}} \qquad \textsc{K-If}\ \frac{\Gamma \vdash \texttt{e}_0 : \mathbb{B} \qquad \Gamma \vdash \tau_1 :: *^{\mathsf{O}} \qquad \Gamma \vdash \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash \texttt{if e}_0 \texttt{ then } \tau_1 \texttt{ else } \tau_2 :: *^{\mathsf{O}}}$$

$$\textsc{K-Match}\ \frac{\Gamma \vdash \texttt{e}_0 : \tau_1' + \tau_2' \qquad \texttt{x} : \tau_1', \Gamma \vdash \tau_1 :: *^{\mathsf{O}} \qquad \texttt{x} : \tau_2', \Gamma \vdash \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash \texttt{match e}_0 \texttt{ with x}\Rightarrow\tau_1 \texttt{|x}\Rightarrow\tau_2 :: *^{\mathsf{O}}} \qquad \textsc{K-Sub}\ \frac{\Gamma \vdash \tau :: \kappa \qquad \kappa \sqsubseteq \kappa'}{\Gamma \vdash \tau :: \kappa'}$$

Figure 3.11. $\lambda_{\texttt{OADT}}$ kinding rules

done when typing the global context. K-Let, K-If and K-Match 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-Match rule is rather permissive in that it does not require the type of discriminee $\texttt{e}_0$ to be completely publicly typed. While it is unclear when a programmer would ever 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 $\lambda_{\texttt{OADT}}$ is dependently typed, T-If and T-Match rely on an implicit motive that is specialized when typing branches[3]. This motive, (τ), has a special free variable $(\texttt{z})$ which stands in for the term being eliminated. The type used for the `then` branch in T-If $(\texttt{[true/z]}\tau)$ concretizes the occurrences of this variable with `true`, for example. The typing rules for oblivious

[3]↑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.

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

$$\textsc{T-Var}\ \frac{\mathtt{x} : \tau \in \Gamma}{\Gamma \vdash \mathtt{x} : \tau} \qquad \textsc{T-Unit}\ \frac{}{\Gamma \vdash () : \mathbb{1}} \qquad \textsc{T-Lit}\ \frac{}{\Gamma \vdash \mathtt{b} : \mathbb{B}} \qquad \textsc{T-Fun}\ \frac{\mathtt{fn}\ \mathtt{x}{:}\tau\ =\ \mathtt{e} \in \Sigma}{\Gamma \vdash \mathtt{x} : \tau}$$

$$\textsc{T-Abs}\ \frac{\mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e} : \tau_2 \qquad \Gamma \vdash \tau_1 :: *}{\Gamma \vdash \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e} : \Pi \mathtt{x}{:}\tau_1, \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash \mathtt{e}_2 : \Pi \mathtt{x}{:}\tau_1, \tau_2 \qquad \Gamma \vdash \mathtt{e}_1 : \tau_1}{\Gamma \vdash \mathtt{e}_2\ \ \mathtt{e}_1 : [\mathtt{e}_1/\mathtt{x}]\tau_2}$$

$$\textsc{T-Let}\ \frac{\Gamma \vdash \mathtt{e}_1 : \tau_1 \qquad \mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e}_2 : \tau_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}\ =\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : [\mathtt{e}_1/\mathtt{x}]\tau_2} \qquad \textsc{T-Pair}\ \frac{\Gamma \vdash \mathtt{e}_1 : \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 : \tau_2}{\Gamma \vdash (\mathtt{e}_1, \mathtt{e}_2) : \tau_1 \times \tau_2}$$

$$\textsc{T-Proj}\ \frac{\Gamma \vdash \mathtt{e} : \tau_1 \times \tau_2}{\Gamma \vdash \pi_b\ \mathtt{e} : \mathtt{ite}(\mathtt{b}, \tau_1, \tau_2)} \qquad \textsc{T-Inj}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{ite}(\mathtt{b}, \tau_1, \tau_2) \qquad \Gamma \vdash \tau_1 + \tau_2 : *}{\Gamma \vdash \iota_b \texttt{<}\tau_1 + \tau_2\texttt{>}\ \mathtt{e} : \tau_1 + \tau_2}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathbb{B} \qquad \Gamma \vdash \mathtt{e}_1 : [\mathtt{true}/\mathtt{z}]\tau \qquad \Gamma \vdash \mathtt{e}_2 : [\mathtt{false}/\mathtt{z}]\tau}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 : [\mathtt{e}_0/\mathtt{z}]\tau}$$

$$\textsc{T-Match}\ \frac{\Gamma \vdash \mathtt{e}_0 : \tau_1 + \tau_2 \qquad \mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e}_1 : [\mathtt{inl}\texttt{<}\tau_1 + \tau_2\texttt{>}\ \mathtt{x}/\mathtt{z}]\tau \qquad \mathtt{x} : \tau_2, \Gamma \vdash \mathtt{e}_2 : [\mathtt{inr}\texttt{<}\tau_1 + \tau_2\texttt{>}\ \mathtt{x}/\mathtt{z}]\tau}{\Gamma \vdash \mathtt{match}\ \mathtt{e}_0\ \mathtt{with}\ \mathtt{x} \Rightarrow \mathtt{e}_1 \,|\, \mathtt{x} \Rightarrow \mathtt{e}_2 : [\mathtt{e}_0/\mathtt{z}]\tau}$$

$$\textsc{T-Fold}\ \frac{\mathtt{data}\ \mathtt{T}\ =\ \tau \in \Sigma \qquad \Gamma \vdash \mathtt{e} : \tau}{\Gamma \vdash \mathtt{fold}\texttt{<}\mathtt{T}\texttt{>}\ \mathtt{e} : \mathtt{T}} \qquad \textsc{T-Unfold}\ \frac{\mathtt{data}\ \mathtt{T}\ =\ \tau \in \Sigma \qquad \Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{unfold}\texttt{<}\mathtt{T}\texttt{>}\ \mathtt{e} : \tau}$$

$$\textsc{T-Sec}\ \frac{\Gamma \vdash \mathtt{e} : \mathbb{B}}{\Gamma \vdash \widehat{\mathbb{B}}\#\mathtt{s}\ \mathtt{e} : \widehat{\mathbb{B}}} \qquad \textsc{T-Mux}\ \frac{\Gamma \vdash \mathtt{e}_0 : \widehat{\mathbb{B}} \qquad \Gamma \vdash \tau ::_*^{\mathsf{O}} \qquad \Gamma \vdash \mathtt{e}_1 : \tau \qquad \Gamma \vdash \mathtt{e}_2 : \tau}{\Gamma \vdash \mathtt{mux}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2 : \tau}$$

$$\textsc{T-OInj}\ \frac{\Gamma \vdash \mathtt{e} : \mathtt{ite}(\mathtt{b}, \tau_1, \tau_2) \qquad \Gamma \vdash \tau_1 \mathbin{\widehat{+}} \tau_2 ::_*^{\mathsf{O}}}{\Gamma \vdash \widehat{\iota}_b \texttt{<}\tau_1 \mathbin{\widehat{+}} \tau_2\texttt{>}\ \mathtt{e} : \tau_1 \mathbin{\widehat{+}} \tau_2} \qquad \textsc{T-OMatch}\ \frac{\Gamma \vdash \mathtt{e}_0 : \tau_1 \mathbin{\widehat{+}} \tau_2 \qquad \Gamma \vdash \tau ::_*^{\mathsf{O}} \qquad \mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e}_1 : \tau \qquad \mathtt{x} : \tau_2, \Gamma \vdash \mathtt{e}_2 : \tau}{\Gamma \vdash \widehat{\mathtt{match}}\ \mathtt{e}_0\ \mathtt{with}\ \mathtt{x} \Rightarrow \mathtt{e}_1 \,|\, \mathtt{x} \Rightarrow \mathtt{e}_2 : \tau}$$

$$\textsc{T-BoxedLit}\ \frac{}{\Gamma \vdash [\mathtt{b}] : \widehat{\mathbb{B}}}$$

$$\textsc{T-BoxedInj}\ \frac{[\iota_b \texttt{<}\widehat{\omega}\texttt{>}\ \hat{\mathtt{v}}] \Leftarrow \widehat{\omega}}{\Gamma \vdash [\iota_b \texttt{<}\widehat{\omega}\texttt{>}\ \hat{\mathtt{v}}] : \widehat{\omega}} \qquad \textsc{T-Conv}\ \frac{\Gamma \vdash \mathtt{e} : \tau \qquad \tau \equiv \tau' \qquad \Gamma \vdash \tau' :: *}{\Gamma \vdash \mathtt{e} : \tau'}$$

Figure 3.12. $\lambda_{\mathtt{OADT}}$ typing rules

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-OMatch) 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-OMatch.

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, $\Sigma \vdash \mathsf{e} \Rrightarrow \mathsf{e}'$, or simply $\mathsf{e} \Rrightarrow \mathsf{e}'$. 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:

$$\Sigma \vdash \tau_1 \equiv \tau_2 \triangleq \exists \tau.\ \Sigma \vdash \tau_1 \Rrightarrow^* \tau \wedge \Sigma \vdash \tau_2 \Rrightarrow^* \tau$$

Parallel reduction also plays an important role in the metatheory of $\lambda_{\texttt{OADT}}$, particularly in the proof of obliviousness (Theorem 3.2.6).

A subset of the parallel reduction rules are shown in Figure 3.13; the remaining rules are similar to the rules in Figure 2.6 (Chapter 2) and can be found in our Coq development; despite their importance in the metatheory of $\lambda_{\texttt{OADT}}$, 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 3.9. As an example, the parallel reduction rule 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 $\lambda_{\texttt{OADT}}$ program, we also check the definitions in the global context using the rules in Figure 3.14. DT-Fun is straightforward: the type ascription needs to be well-kinded

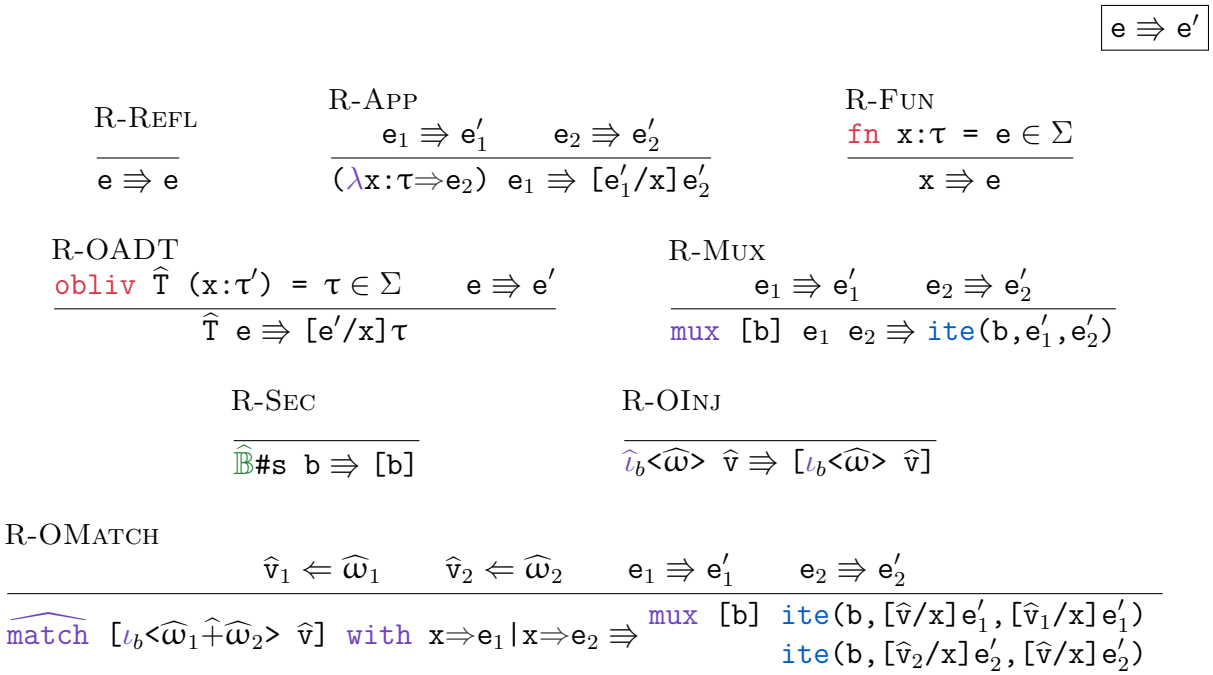


Figure 3.13. Subset of $\lambda_{\texttt{OADT}}$ parallel reduction rules

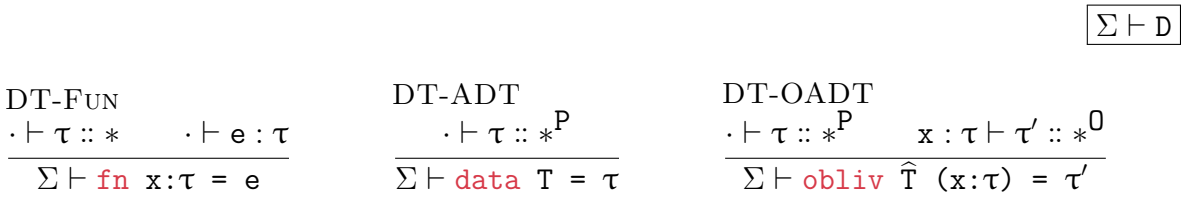


Figure 3.14. $\lambda_{\texttt{OADT}}$ global definition typing rules

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.

3.2.4 Type Safety and Obliviousness

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

Theorem 3.2.1 (Progress)**.** *If* $\cdot \vdash \mathsf{e} : \tau$*, then either* $\mathsf{e} \longrightarrow \mathsf{e}'$ *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-OMatch 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 3.2.2 (Preservation)**.** *If* $\Gamma \vdash \mathsf{e} : \tau$*, and* $\mathsf{e} \longrightarrow \mathsf{e}'$*, then* $\Gamma \vdash \mathsf{e}' : \tau$*.*
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-Match cases. Instead, we show that the step relation refines parallel reduction and then prove preservation for the more general relation.

Lemma 3.2.3 (Preservation for parallel reduction)**.** *If* $\Gamma \vdash \mathsf{e} : \tau$*, and* $\mathsf{e} \Rrightarrow \mathsf{e}'$*, then* $\Gamma \vdash \mathsf{e}' : \tau$*.*
If $\Gamma \vdash \tau :: \kappa$ *and* $\tau \Rrightarrow \tau'$*, then* $\Gamma \vdash \tau' :: \kappa$*.*

The proof of Lemma 3.2.3 depends on two additional lemmas. The first is a regularity lemma needed for the kinding constraints used by several typing rules.

Lemma 3.2.4 (Regularity)**.** *If* $\Gamma \vdash \mathsf{e} : \tau$*, then* $\Gamma \vdash \tau :: \kappa$ *for some* κ*.*

The second is that parallel reduction is confluent.

Lemma 3.2.5 (Confluence of parallel reduction)**.** *If* $\mathsf{e} \Rrightarrow^* \mathsf{e}_1$ *and* $\mathsf{e} \Rrightarrow^* \mathsf{e}_2$*, then there exists* e' *such that* $\mathsf{e}_1 \Rrightarrow^* \mathsf{e}'$ *and* $\mathsf{e}_2 \Rrightarrow^* \mathsf{e}'$*.*

Interestingly, the regular call-by-value semantics of $\lambda_{\texttt{OADT}}$ are not confluent, thanks to a combination of the (limited) nondeterminism in S-OMatch and nontermination. Observe

that S-OMatch 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 $\widehat{\texttt{match}}$ parallel reduces to a mux expression, however, R-Mux will immediately discard the "wrong" branch, forcing both choices to converge within one step.

Obliviousness

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

Definition 3.2.1 (Indistinguishability)**.** We say two expressions are *indistinguishable*, denoted by $\texttt{e} \approx \texttt{e}'$, if

1. they are both oblivious boolean values: $\texttt{[b]} \approx \texttt{[b}'\texttt{]}$, or
2. they are both oblivious injections with the same type: $\texttt{[}\iota_b\texttt{<}\widehat{\omega}\texttt{> v]} \approx \texttt{[}\iota_{b'}\texttt{<}\widehat{\omega}\texttt{> v}'\texttt{]}$, 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 λx y⇒x+y and λx y⇒y+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 λx y⇒x+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 $\lambda_{\texttt{OADT}}$:

Theorem 3.2.6 (Obliviousness)**.** *If* $\mathsf{e}_1 \approx \mathsf{e}_2$ *and* $\cdot \vdash \mathsf{e}_1 : \tau_1$ *and* $\cdot \vdash \mathsf{e}_2 : \tau_2$*, then*

1. $\mathsf{e}_1 \longrightarrow^n \mathsf{e}_1'$ *if and only if* $\mathsf{e}_2 \longrightarrow^n \mathsf{e}_2'$ *for some* e_2'*.*

2. *if* $\mathsf{e}_1 \longrightarrow^n \mathsf{e}_1'$ *and* $\mathsf{e}_2 \longrightarrow^n \mathsf{e}_2'$*, then* $\mathsf{e}_1' \approx \mathsf{e}_2'$*.*

We write $\mathsf{e} \longrightarrow^n \mathsf{e}'$ to mean e reduces to 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 $\lambda_{\texttt{OADT}}$ 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 [21, 27], in that different private (i.e., high-security) inputs do not leak any information via this public channel.

The proof of Theorem 3.2.6 is by induction on the derivation of $\mathsf{e}_1 \longrightarrow^n \mathsf{e}_1'$. 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 3.2.7. *If* $\Gamma \vdash \mathsf{v} : \tau$ *and* $\Gamma \vdash \mathsf{v}' : \tau$*, and* $\Gamma \vdash \tau :: *^{\mathsf{O}}$*, then* $\mathsf{v} \approx \mathsf{v}'$*.*

Lemma 3.2.8. *If* $\mathsf{v} \approx \mathsf{v}'$*,* $\Gamma \vdash \mathsf{v} : \tau$*,* $\Gamma \vdash \mathsf{v}' : \tau'$*, and* $\Gamma \vdash \tau :: *^{\mathsf{O}}$*, then* $\tau \equiv \tau'$*.*

Lemma 3.2.7 states that all values of the same oblivious type are indistinguishable, and Lemma 3.2.8 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 3.2.3:

Lemma 3.2.9. *If* $\tau \equiv \tau'$, $\Gamma \vdash \tau :: *^{\mathsf{O}}$, *and* $\Gamma \vdash \tau' :: *$, *then* $\Gamma \vdash \tau' :: *^{\mathsf{O}}$.

In practice, well-typed $\lambda_{\mathtt{OADT}}$ programs are functions that take arguments of oblivious types, such as `lookupO` from Figure 3.7. 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 3.2.7. 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 $\lambda_{\mathtt{OADT}}$ programs:

Corollary 3.2.10. *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. $\mathtt{[v_1/x]e} \longrightarrow^n \mathtt{e}_1$ *if and only if* $\mathtt{[v_2/x]e} \longrightarrow^n \mathtt{e}_2$ *for some* $\mathtt{e}_2$.

2. $\mathtt{[v_1/x]e} \longrightarrow^n \mathtt{e}_1$ *and* $\mathtt{[v_2/x]e} \longrightarrow^n \mathtt{e}_2$ *implies that* $\mathtt{e}_1$ *and* $\mathtt{e}_2$ *are indistinguishable, i.e.,* $\mathtt{e}_1 \approx \mathtt{e}_2$.

3.3 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 $\lambda_{\mathtt{OADT}}$, a core calculus for encoding oblivious programs over oblivious algebraic data types. $\lambda_{\mathtt{OADT}}$ 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. 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.

4. TAPE SEMANTICS: DYNAMIC ENFORCEMENT OF POLICIES

While $\lambda_{\texttt{OADT}}$ provides a foundation for implementing secure computation that involves structured data and complex policies, writing secure programs directly in $\lambda_{\texttt{OADT}}$ is challenging, especially for programmers who are not well-versed in dependent types or information flow control. This chapter introduces a semantics-based approach to enforce privacy policies dynamically. This novel execution model, called *tape semantics*, allows programs to include unsafe computations and repairs these unsafe computations at runtime. Tape semantics decouples privacy and programmatic concerns, enabling a sort of modular design that allows programmers to implement the functionality of their secure applications in a standard way. Using our language, clients can write a single function over private data, and then build an equivalent oblivious computation over some public view (i.e., policy) 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 chapter presents the following contributions:

- To enable both a more pleasant programming experience and more modular programs, we develop an extension of $\lambda_{\texttt{OADT}}$, dubbed $\lambda_{\texttt{OADT}\textbf{+}}$, which is equipped with a novel semantics which enables creating, from one single public program, oblivious programs with different public views.

- We present a reference semantics and connect it to tape semantics. The correspondence between tape semantics and a more standard semantics allows us to reason about programs in $\lambda_{\texttt{OADT}\textbf{+}}$ using traditional mindset and methods.

- To further reduce the user burden, we develop an algorithm to derive secure implementations in $\lambda_{\texttt{OADT}\textbf{+}}$ from public programs and their privacy policies as type signatures.

Similar to $\lambda_{\texttt{OADT}}$, $\lambda_{\texttt{OADT}\textbf{+}}$ and its metatheory have been mechanized in the Coq proof assistant [48].

4.1 Overview

Recall the secure implementation of `lookup0` from Figure 3.7 in Chapter 3. While this implementation is guaranteed to be secure, it is quite far from the "standard" implementation of `lookup` in Figure 3.1, 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 $\lambda_{\texttt{OADT}}$ program, *as long as* it is applied to public data. This observation suggests the implementation of $\widehat{\texttt{lookup}}$ sketched in Figure 4.1, which simply converts its private inputs to public versions, applies `lookup` to those arguments, and converts the result back to an oblivious value. Recall that these conversions are called section and retraction (Section 3.1.1). From a cryptographic perspective, we "decrypt" the secure inputs using retraction functions (e.g., $\widehat{\mathbb{Z}}$`#r` and $\widehat{\texttt{tree}}$`#r`), and "encrypt" the computed result using section functions (e.g., $\widehat{\mathbb{B}}$`#s`). 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 $\lambda_{\texttt{OADT}}$'s type system as insecure.

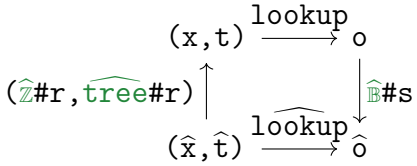


Figure 4.1. Sketch of a secure `lookup` function

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 $\lambda_{\texttt{OADT}}$, called $\lambda_{\texttt{OADT}\blacksquare}$, that allows implementations that follow the recipe sketched in Figure 4.1 *without* compromising obliviousness. Our key idea is to have the semantics of $\lambda_{\texttt{OADT}\blacksquare}$ 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.

To understand how this works, consider the execution trace of the simple $\lambda_{\texttt{OADT}\blacksquare}$ program shown in Figure 4.2a. The new conditional $\widehat{\texttt{if}}$ is similar to `mux` in $\lambda_{\texttt{OADT}}$, but it allows

```
tape (if (if [true]                tape (if [true]
         then true                       then if true then [5]
         else false)        ⟶                         else [4]
     then [5]                           else if false then [5]
     else [4])                                  else [4])
```

```
⟶* tape (if [true] then [5]
                   else [4])   ⟶  mux [true] [5] [4]  ⟶  [5]
```

(a) if inside if

```
tape ((if [true]                       tape (if [true]
      then (λx ⇒ x + [1])      ⟶             then (λx ⇒ x + [1]) [4]
      else (λx ⇒ x)) [4])                    else (λx ⇒ x) [4])
```

```
⟶* tape (if [true] then [5]
                   else [4])   ⟶  mux [true] [5] [4]  ⟶  [5]
```

(b) if inside application

```
tape (ℤ#s (ℤ#r [2] + ℤ#r [3]))  ⟶  tape (ℤ#s (ℤ#r ([2] + [3])))
⟶  tape (ℤ#s (ℤ#r [5]))  ⟶  tape [5]  ⟶  [5]
```

(c) retraction of integer

Figure 4.2. Example $\lambda_{\text{OADT}\blacksquare}$ execution traces

non-oblivious branches. Note that this if would leak the value of its private condition if it was evaluated using the semantics of mux. Similar leaks occur for any if expression whose branches can evaluate to a public value. The idea behind the semantics of $\lambda_{\text{OADT}\blacksquare}$ is straightforward: since if only leaks information when it is evaluated, we will simply not do that! Rather, the surrounding tape annotation ensures the expression will be *eventually oblivious*, and tells $\lambda_{\text{OADT}\blacksquare}$ to defer reducing 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 if are evaluated to oblivious values, it can be securely reduced to a mux to produce the final result. Note that

```
fn tree#s {⊥} (k : ℕ)⊥ (t : tree)⊤ : tree k =
  if k = 0
  then ()
  else tape (match t with
             | Leaf ⇒ inl ()
             | Node x tl tr ⇒
               inr (tape (ℤ#s x, tree#s tl (k−1), tree#s tr (k−1))))

fn tree#r {⊤} (k : ℕ)⊥ : (tree k)⊥ → tree =
  if k = 0
  then λ_ ⇒ Leaf
  else λt ⇒ match t of
             | inl _ ⇒ Leaf
             | inr (x, tl, tr) ⇒
               Node (ℤ#r x) (tree#r (k−1) tl) (tree#r (k−1) tr)

fn lookup {⊥} (k : ℕ)⊥ (x : ℤ)⊥ (t : tree k)⊥ : 𝔹 =
  tape (𝔹#s (lookup (ℤ#r x) (tree#r k t)))
```

Figure 4.3. Oblivious lookup function in $\lambda_{\text{OADT}\blacksquare}$

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 $\lambda_{\text{OADT}\blacksquare}$: 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 4.2b shows. Figure 4.2c includes an example of a potential leak of oblivious integers via the $\hat{\mathbb{Z}}$`#r` operation, a leak that is ultimately patched using $\hat{\mathbb{Z}}$`#s`. 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 $\hat{\mathbb{Z}}$`#s` and $\hat{\mathbb{Z}}$`#r` 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 4.3 shows the section and retraction functions for $\widehat{\texttt{tree}}$, along with a version of $\widehat{\texttt{lookup}}$ implemented using the recipe from Figure 4.1. While the section function is not used in $\widehat{\texttt{lookup}}$, it is needed for functions that return an oblivious tree. Function definitions in $\lambda_{\texttt{OADT}\blacksquare}$ require an 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 4.2 discusses how the type system of $\lambda_{\texttt{OADT}\blacksquare}$ 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.

This strategy of decoupling program logic and privacy policies enjoys multiple benefits. First, the core program logic is easier to read, write and reason about, because it is simply a normal functional program, just like `lookup`. Second, these core functions are agnostic to a particular security policy. To share the spine of the tree, we only need to choose a different $\widehat{\texttt{tree}}$`#r` and $\widehat{\texttt{tree}}$`#s`; `lookup` itself remains unchanged. This frees users from writing different versions of the same function for different security policies. Third, this approach allows users to experiment and trade off between performance and security guarantee. Sharing the exact spine of the tree will result in better performance, for example, if both parties agree to this policy.

4.2 $\lambda_{\texttt{OADT}\blacksquare}$, Formally

This section formalizes $\lambda_{\texttt{OADT}\blacksquare}$, an extension to $\lambda_{\texttt{OADT}}$ that permits implementations in the vein of Figure 4.1.

4.2.1 Syntax

The extended syntax of $\lambda_{\texttt{OADT}\blacksquare}$ is shown in Figure 4.4. These extensions permit $\lambda_{\texttt{OADT}\blacksquare}$ expressions that *potentially* leak information *locally*, as long as they can eventually be repaired by the surrounding context. The new $\widehat{\texttt{if}}$ operation is similar to `mux`, but its branches are

e,τ	::=		Extended Expressions:
	|	...	
	|	$\widehat{\texttt{if}}$ e then e else e	oblivious leaky conditional
	|	tape e	tape operation
	|	let x:$_l$τ = e in e	let binding with leakage label
	|	λx:$_l$τ⇒e | Πx:$_l$τ,τ	function and function types with leakage label
D	::=		Extended Global Definitions:
	|	...	
	|	fn x:$_l$τ = e	(recursive) function definition with leakage label
l	::=	⊤ | ⊥	Leakage Label

Figure 4.4. $\lambda_{\texttt{OADT}\blacksquare}$ syntax

permitted to be non-oblivious, causing a potential leak if $\widehat{\texttt{if}}$ is evaluated naively. The new tape annotation acts as a boundary for potential leaks, and is used to ensure that they never occur during execution, as Section 4.2.2 will discuss in more detail. Finally, $\lambda_{\texttt{OADT}\blacksquare}$ updates the syntax for let bindings, anonymous functions, function types and function definitions with a *leakage label*. A leakage label is either ⊤ or ⊥, and signals either the presence or the absence of a potential leak, respectively.

4.2.2 Semantics

The semantics of $\lambda_{\texttt{OADT}\blacksquare}$ are an extension of the semantics of $\lambda_{\texttt{OADT}}$. Figure 4.5 shows the new and updated rules; the rest are identical to the rules in Figure 3.9. 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 $\lambda_{\texttt{OADT}}$ 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 4.5) 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

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

S-OMatch

$$\frac{\hat{\mathtt{v}}_1 \Leftarrow \widehat{\omega}_1 \qquad \hat{\mathtt{v}}_2 \Leftarrow \widehat{\omega}_2}{\widehat{\mathtt{match}}\ [\iota_b\mathtt{<}\widehat{\omega}_1 \hat{+} \widehat{\omega}_2\mathtt{>}\ \hat{\mathtt{v}}]\ \mathtt{with}\ \mathtt{x}\Rightarrow\mathtt{e}_1\,|\,\mathtt{x}\Rightarrow\mathtt{e}_2 \longrightarrow \begin{array}{l}\widehat{\mathtt{if}}\ [\mathtt{b}]\ \mathtt{then}\ \mathtt{ite}(\mathtt{b},[\hat{\mathtt{v}}/\mathtt{x}]\mathtt{e}_1,[\hat{\mathtt{v}}_1/\mathtt{x}]\mathtt{e}_1)\\ \mathtt{else}\ \mathtt{ite}(\mathtt{b},[\hat{\mathtt{v}}_2/\mathtt{x}]\mathtt{e}_2,[\hat{\mathtt{v}}/\mathtt{x}]\mathtt{e}_2)\end{array}}$$

S-OIf

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

S-TapeOIf

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

S-TapeOVal

$\hat{\mathtt{v}}$ is oblivious value but not pair

$$\frac{}{\mathtt{tape}\ \hat{\mathtt{v}} \longrightarrow \hat{\mathtt{v}}}$$

S-TapePair

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

Weak Values

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

Evaluation Contexts

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

Leaky Contexts

$$\begin{array}{lcl}\hat{\mathcal{E}} & ::= & \\ & \mid & \square\ \mathtt{v}\\ & \mid & \pi_b\ \square\\ & \mid & \mathtt{if}\ \square\ \mathtt{then}\ \mathtt{e}\ \mathtt{else}\ \mathtt{e}\\ & \mid & \mathtt{match}\ \square\ \mathtt{with}\ \mathtt{x}\Rightarrow\mathtt{e}\,|\,\mathtt{x}\Rightarrow\mathtt{e}\\ & \mid & \hat{\mathbb{B}}\mathtt{\#s}\ \square\\ & \mid & \mathtt{unfold<T>}\ \square\end{array}$$

Figure 4.5. $\lambda_{\mathtt{OADT}\boldsymbol{+}}$ semantics

make progress; pushing `fold` into $\widehat{\mathtt{if}}$ in the expression `fold<tree>` ($\widehat{\mathtt{if}}$ `[true] ...`) does not gain us anything, for example, since the expression is already a weak value. Figure 4.5 defines the *leaky contexts* ($\hat{\mathcal{E}}$) that can be distributed through $\widehat{\mathtt{if}}$. 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 $\lambda_{\mathtt{OADT}\boldsymbol{+}}$, for similar reasons to the fold example from above. The semantics of $\widehat{\mathtt{match}}$ are also updated to allow potential leaks, with S-OMatch now evaluating to $\widehat{\mathtt{if}}$ instead of `mux`.

The last three rules in Figure 4.5 show how to evaluate `tape` annotations. The key idea is to use `tape` as a signal that the context surrounding an $\widehat{\mathtt{if}}$ expression has been sufficiently distributed to prevent leaks. Mechanically, whenever a `tape` annotation is applied to $\widehat{\mathtt{if}}$

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:

```
      tape (B̂#s (if [true]          tape (if [true]
                then false    ⟶          then (B̂#s false)
                else true))               else (B̂#s true))

      tape (if [true]            mux [true]
⟶*          then [false]   ⟶         (tape [false])
            else [true])             (tape [true])

⟶*  mux [true] [false] [true]  ⟶  [false]
```

After applying S-OIf to distribute the surrounding boolean section $\widehat{\mathbb{B}}$`#s`, 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.

4.2.3 Type System

The typing judgment of $\lambda_{\texttt{OADT}\blacksquare}$ now includes a leakage label for the typed expression: $\Gamma \vdash \texttt{e} :_{\ell} \tau$, as do entries in typing contexts Γ. Figure 4.6 shows a subset of the typing rules of $\lambda_{\texttt{OADT}\blacksquare}$; the omitted rules are copies of those from $\lambda_{\texttt{OADT}}$ with straightforward leakage labels annotations. As mentioned in Section 4.2.1, 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 [27, 29], where type-based information flow control is used to enforce noninterference between high- and low- security information. In $\lambda_{\texttt{OADT}\blacksquare}$, expressions with $\top$ labels should not influence expressions with $\bot$ labels. In order to minimize the extension to $\lambda_{\texttt{OADT}}$, 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

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

$$\textsc{T-Var}\ \frac{\mathtt{x} :_l \tau \in \Gamma}{\Gamma \vdash \mathtt{x} :_l \tau} \qquad \textsc{T-Unit}\ \frac{}{\Gamma \vdash () :_\bot \mathbb{1}} \qquad \textsc{T-Pair}\ \frac{\Gamma \vdash \mathtt{e}_1 :_{l_1} \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :_{l_2} \tau_2 \qquad l = l_1 \sqcup l_2}{\Gamma \vdash (\mathtt{e}_1, \mathtt{e}_2) :_l \tau_1 \times \tau_2} \qquad \textsc{T-Proj}\ \frac{\Gamma \vdash \mathtt{e} :_l \tau_1 \times \tau_2}{\Gamma \vdash \pi_b\ \mathtt{e} :_l \mathtt{ite}(\mathtt{b}, \tau_1, \tau_2)}$$

$$\textsc{T-Abs}\ \frac{\mathtt{x} :_{l_1} \tau_1, \Gamma \vdash \mathtt{e} :_{l_2} \tau_2 \qquad \Gamma \vdash \tau_1 :: *}{\Gamma \vdash \lambda \mathtt{x}:_{l_1} \tau_1 \Rightarrow \mathtt{e} :_{l_2} \Pi \mathtt{x}:_{l_1} \tau_1, \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash \mathtt{e}_2 :_{l_2} \Pi \mathtt{x}:_{l_1} \tau_1, \tau_2 \qquad \Gamma \vdash \mathtt{e}_1 :_{l_1} \tau_1}{\Gamma \vdash \mathtt{e}_2\ \mathtt{e}_1 :_{l_2} [\mathtt{e}_1/\mathtt{x}]\tau_2}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash \mathtt{e}_0 :_\bot \mathbb{B} \qquad l = l_1 \sqcup l_2 \qquad \Gamma \vdash \mathtt{e}_1 :_{l_1} [\mathtt{true}/\mathtt{z}]\tau \qquad \Gamma \vdash \mathtt{e}_2 :_{l_2} [\mathtt{false}/\mathtt{z}]\tau}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 :_l [\mathtt{e}_0/\mathtt{z}]\tau} \qquad \textsc{T-IfNoDep}\ \frac{\Gamma \vdash \mathtt{e}_0 :_{l_0} \mathbb{B} \qquad l = l_0 \sqcup l_1 \sqcup l_2 \qquad \Gamma \vdash \mathtt{e}_1 :_{l_1} \tau \qquad \Gamma \vdash \mathtt{e}_2 :_{l_2} \tau}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 :_l \tau}$$

$$\textsc{T-Match}\ \frac{\Gamma \vdash \mathtt{e}_0 :_\bot \tau_1 + \tau_2 \qquad l = l_1 \sqcup l_2 \qquad \mathtt{x} :_\bot \tau_1, \Gamma \vdash \mathtt{e}_1 :_{l_1} [\mathtt{inl}\texttt{<}\tau_1+\tau_2\texttt{>}\ \mathtt{x}/\mathtt{z}]\tau \qquad \mathtt{x} :_\bot \tau_2, \Gamma \vdash \mathtt{e}_2 :_{l_2} [\mathtt{inr}\texttt{<}\tau_1+\tau_2\texttt{>}\ \mathtt{x}/\mathtt{z}]\tau}{\Gamma \vdash \mathtt{match}\ \mathtt{e}_0\ \mathtt{with}\ \mathtt{x} \Rightarrow \mathtt{e}_1 \mid \mathtt{x} \Rightarrow \mathtt{e}_2 :_l [\mathtt{e}_0/\mathtt{z}]\tau}$$

$$\textsc{T-MatchNoDep}\ \frac{\Gamma \vdash \mathtt{e}_0 :_{l_0} \tau_1 + \tau_2 \qquad l = l_0 \sqcup l_1 \sqcup l_2 \qquad \mathtt{x} :_{l_0} \tau_1, \Gamma \vdash \mathtt{e}_1 :_{l_1} \tau \qquad \mathtt{x} :_{l_0} \tau_2, \Gamma \vdash \mathtt{e}_2 :_{l_2} \tau}{\Gamma \vdash \mathtt{match}\ \mathtt{e}_0\ \mathtt{with}\ \mathtt{x} \Rightarrow \mathtt{e}_1 \mid \mathtt{x} \Rightarrow \mathtt{e}_2 :_l \tau} \qquad \textsc{T-Conv}\ \frac{\Gamma \vdash \mathtt{e} :_l \tau \qquad \tau \equiv \tau' \qquad \Gamma \vdash \tau' :: * \qquad l \sqsubseteq l'}{\Gamma \vdash \mathtt{e} :_{l'} \tau'}$$

$$\textsc{T-Mux}\ \frac{\Gamma \vdash \mathtt{e}_0 :_\bot \widehat{\mathbb{B}} \qquad \Gamma \vdash \tau :: *^0 \qquad \Gamma \vdash \mathtt{e}_1 :_\bot \tau \qquad \Gamma \vdash \mathtt{e}_2 :_\bot \tau}{\Gamma \vdash \mathtt{mux}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2 :_\bot \tau} \qquad \textsc{T-OInj}\ \frac{\Gamma \vdash \mathtt{e} :_\bot \mathtt{ite}(\mathtt{b}, \tau_1, \tau_2) \qquad \Gamma \vdash \tau_1 \widehat{+} \tau_2 :: *^0}{\Gamma \vdash \widehat{\iota}_b\texttt{<}\tau_1 \widehat{+} \tau_2\texttt{>}\ \mathtt{e} :_\bot \tau_1 \widehat{+} \tau_2}$$

$$\textsc{T-OIf}\ \frac{\Gamma \vdash \mathtt{e}_0 :_\bot \widehat{\mathbb{B}} \qquad \Gamma \vdash \mathtt{e}_1 :_{l_1} \tau \qquad \Gamma \vdash \mathtt{e}_2 :_{l_2} \tau}{\Gamma \vdash \widehat{\mathtt{if}}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 :_\top \tau} \qquad \textsc{T-OMatch}\ \frac{\Gamma \vdash \mathtt{e}_0 :_\bot \tau_1 \widehat{+} \tau_2 \qquad \mathtt{x} :_\bot \tau_1, \Gamma \vdash \mathtt{e}_1 :_{l_1} \tau \qquad \mathtt{x} :_\bot \tau_2, \Gamma \vdash \mathtt{e}_2 :_{l_2} \tau}{\Gamma \vdash \widehat{\mathtt{match}}\ \mathtt{e}_0\ \mathtt{with}\ \mathtt{x} \Rightarrow \mathtt{e}_1 \mid \mathtt{x} \Rightarrow \mathtt{e}_2 :_\top \tau}$$

$$\textsc{T-Tape}\ \frac{\Gamma \vdash \mathtt{e} :_l \tau \qquad \Gamma \vdash \tau :: *^0}{\Gamma \vdash \mathtt{tape}\ \mathtt{e} :_\bot \tau}$$

Figure 4.6. Selected $\lambda_{\mathtt{OADT}\textbf{+}}$ typing rules

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 `(λx:⊤𝔹⇒𝔹#s x) (if [true] then true else false)` and `(λx:⊤𝔹⇒𝔹#s x) true` are well-typed expressions.

$\lambda_{\texttt{OADT}\blacksquare}$ 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 `match`, T-Match and T-MatchNoDep, 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 ($\widehat{\texttt{if}}$). The former is already safe, and the latter can be repaired by S-TapeOIf. The $\bot$ label in the rule captures the idea that `tape` safely repairs a local leak, such that the surrounding

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

$$\textsc{K-OADT}\ \frac{\texttt{obliv}\ \widehat{\texttt{T}}\ (\texttt{x}{:}\tau)\ =\ \tau' \in \Sigma \qquad \Gamma \vdash \texttt{e} :_{\bot} \tau}{\Gamma \vdash \widehat{\texttt{T}}\ \texttt{e} :: *^{\mathsf{O}}} \qquad \textsc{K-If}\ \frac{\Gamma \vdash \texttt{e}_0 :_{\bot} \mathbb{B} \qquad \Gamma \vdash \tau_1 :: *^{\mathsf{O}} \qquad \Gamma \vdash \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash \texttt{if}\ \texttt{e}_0\ \texttt{then}\ \tau_1\ \texttt{else}\ \tau_2 :: *^{\mathsf{O}}}$$

$$\textsc{K-Match}\ \frac{\Gamma \vdash \texttt{e}_0 :_{\bot} \tau_1' + \tau_2' \qquad \texttt{x} :_{\bot} \tau_1', \Gamma \vdash \tau_1 :: *^{\mathsf{O}} \qquad \texttt{x} :_{\bot} \tau_2', \Gamma \vdash \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash \texttt{match}\ \texttt{e}_0\ \texttt{with}\ \texttt{x} \Rightarrow \tau_1 \,|\, \texttt{x} \Rightarrow \tau_2 :: *^{\mathsf{O}}}$$

Figure 4.7. Selected $\lambda_{\texttt{OADT}\blacksquare}$ kinding rules

computation can treat it as non-leaky. The rules for $\widehat{\texttt{if}}$ and $\widehat{\texttt{match}}$ 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.

Figure 4.7 shows the updated kinding rules for $\lambda_{\texttt{OADT}\blacksquare}$; the other kinding rules are identical to those in Figure 3.11. 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 `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 $\widehat{\mathbb{B}}\texttt{\#s}$ 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 4.8 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 $\lambda_{\texttt{OADT}\blacksquare}$ program is well-typed if the global context is well-typed (the updated typing rules for the global context

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

R-OIfCtx

$$\frac{\mathsf{e}_1 \Rrightarrow \mathsf{e}_1' \qquad \mathsf{e}_2 \Rrightarrow \mathsf{e}_2' \qquad \widehat{\mathcal{E}} \Rrightarrow \widehat{\mathcal{E}}'}{\widehat{\mathcal{E}}[\widehat{\texttt{if}}\ \texttt{[b]}\ \texttt{then}\ \mathsf{e}_1\ \texttt{else}\ \mathsf{e}_2] \Rrightarrow \widehat{\texttt{if}}\ \texttt{[b]}\ \texttt{then}\ \widehat{\mathcal{E}}'[\mathsf{e}_1']\ \texttt{else}\ \widehat{\mathcal{E}}'[\mathsf{e}_2']}$$

R-TapeOIf

$$\frac{\mathsf{e}_1 \Rrightarrow \mathsf{e}_1' \qquad \mathsf{e}_2 \Rrightarrow \mathsf{e}_2'}{\texttt{tape}\ (\widehat{\texttt{if}}\ \texttt{[b]}\ \texttt{then}\ \mathsf{e}_1\ \texttt{else}\ \mathsf{e}_2) \Rrightarrow \texttt{mux}\ \texttt{[b]}\ (\texttt{tape}\ \mathsf{e}_1')\ (\texttt{tape}\ \mathsf{e}_2')}$$

R-TapePair

$$\frac{\mathsf{e}_1 \Rrightarrow \mathsf{e}_1' \qquad \mathsf{e}_2 \Rrightarrow \mathsf{e}_2'}{\texttt{tape}\ (\mathsf{e}_1,\mathsf{e}_2) \Rrightarrow (\texttt{tape}\ \mathsf{e}_1',\texttt{tape}\ \mathsf{e}_2')}$$

R-TapeOVal

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

R-OIf

$$\frac{\mathsf{e}_1 \Rrightarrow \mathsf{e}_1' \qquad \mathsf{e}_2 \Rrightarrow \mathsf{e}_2'}{\widehat{\texttt{if}}\ \texttt{[b]}\ \texttt{then}\ \mathsf{e}_1\ \texttt{else}\ \mathsf{e}_2 \Rrightarrow \texttt{ite}(\mathsf{b},\mathsf{e}_1',\mathsf{e}_2')}$$

Figure 4.8. Selected $\lambda_{\texttt{OADT}\blacksquare}$ parallel reduction rules

are trivial) and the expression is well-typed with $\bot$ label. The latter restriction ensures that all potential leaks in a $\lambda_{\texttt{OADT}\blacksquare}$ program are eventually repaired.

4.2.4 Type Safety and Obliviousness

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

Theorem 4.2.1 (Progress)**.** *If* $\cdot \vdash \mathsf{e} :_{\bot} \tau$*, then either* $\mathsf{e} \longrightarrow \mathsf{e}'$ *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.*

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

The proof of this stronger lemma proceeds similarly to the proof of progress for $\lambda_{\texttt{OADT}}$, 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 3.2.9, so the proof of progress for $\lambda_{\texttt{OADT+}}$ 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 4.2.3 (Preservation)**.** *If* $\Gamma \vdash \mathsf{e} :_l \tau$*, and* $\mathsf{e} \longrightarrow \mathsf{e}'$*, then* $\Gamma \vdash \mathsf{e}' :_l \tau$*.*

If $\Gamma \vdash \tau :: \kappa$ *and* $\tau \longrightarrow \tau'$*, then* $\Gamma \vdash \tau' :: \kappa$*.*

Theorem 4.2.4 (Obliviousness)**.** *If* $\mathsf{e}_1 \approx \mathsf{e}_2$ *and* $\cdot \vdash \mathsf{e}_1 :_{l_1} \tau_1$ *and* $\cdot \vdash \mathsf{e}_2 :_{l_2} \tau_2$*, then*

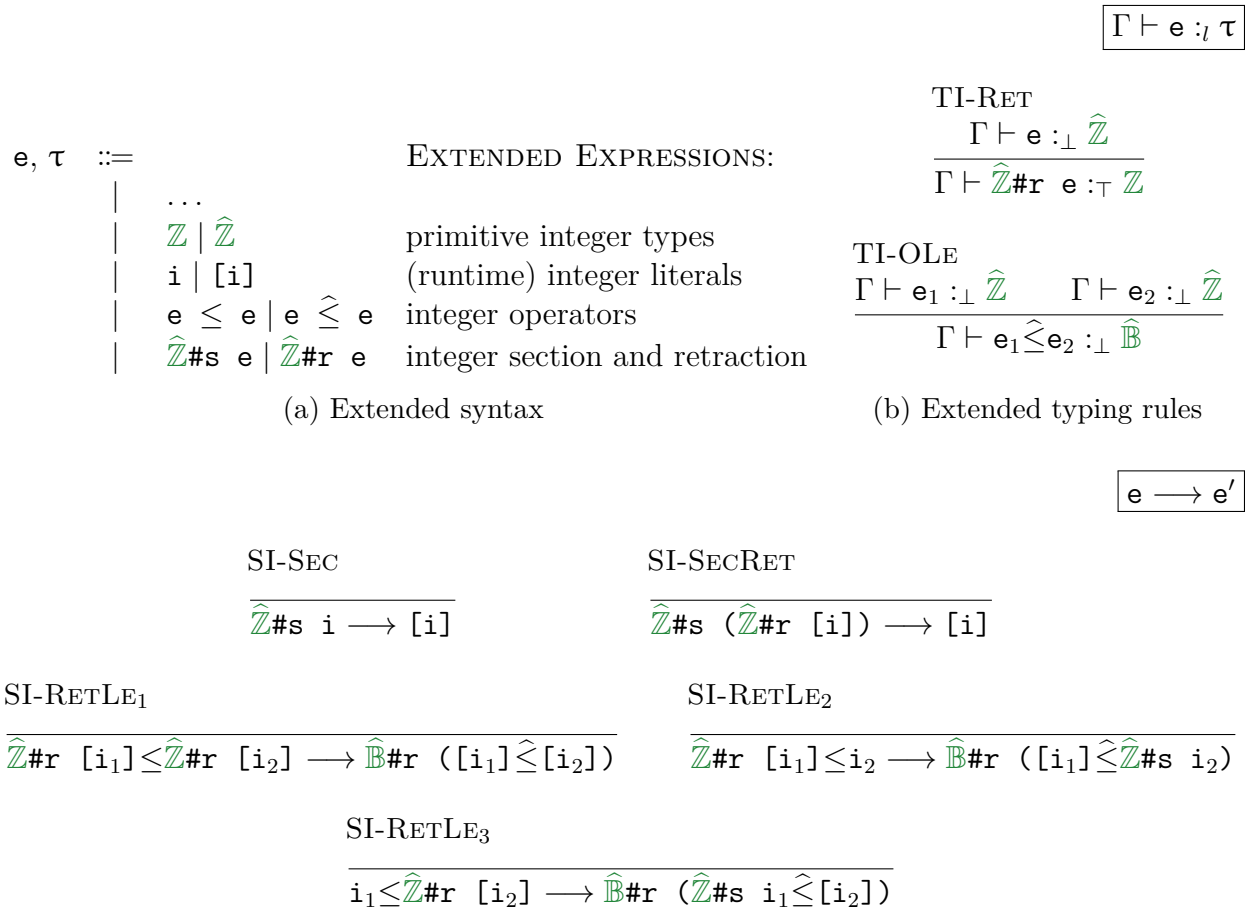
1. $\mathsf{e}_1 \longrightarrow^n \mathsf{e}'_1$ *if and only if* $\mathsf{e}_2 \longrightarrow^n \mathsf{e}'_2$ *for some* e'_2*.*

2. *if* $\mathsf{e}_1 \longrightarrow^n \mathsf{e}'_1$ *and* $\mathsf{e}_2 \longrightarrow^n \mathsf{e}'_2$*, then* $\mathsf{e}'_1 \approx \mathsf{e}'_2$*.*

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

4.3 Extending $\lambda_{\texttt{OADT+}}$

This section considers how additional base types might be added to the core calculus of $\lambda_{\texttt{OADT+}}$, using fixed-width integers as an example. Figure 4.9 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 ($\widehat{\mathbb{Z}}$`#s`) and retraction ($\widehat{\mathbb{Z}}$`#r`) operations for integers are also added; both have similar semantics to their



(a) Extended syntax

(b) Extended typing rules

(c) Extended semantics

Figure 4.9. A subset of extended language for fixed-width integers

boolean counterparts[4]. $\widehat{\mathbb{Z}}\mathtt{\#r}$ always introduces a potential leak, and $\widehat{\mathbb{Z}}\mathtt{\#r\ \hat{v}}$ 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 $\widehat{\mathbb{B}}\mathtt{\#r}$ 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 values first. The semantics of other operators can be defined

[4]↑Although $\lambda_{\text{OADT}\blacksquare}$ does not include boolean retraction $\widehat{\mathbb{B}}\mathtt{\#r}$ as a primitive, it is easily defined in terms of $\widehat{\mathtt{if}}$: $\widehat{\mathbb{B}}\mathtt{\#r\ e} \triangleq \widehat{\mathtt{if}}\ \mathtt{e\ then\ true\ else\ false}$.

through similar uses of section and retraction functions. As an example, integer addition returns an integer instead of boolean, so we apply $\widehat{\mathbb{Z}}\texttt{\#r}$ to the result of oblivious addition. If a leaky integer expression is used in a well-typed context, then $\widehat{\mathbb{Z}}\texttt{\#r}$ will eventually meet $\widehat{\mathbb{Z}}\texttt{\#s}$ 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 $\lambda_{\texttt{OADT}\blacksquare}$. The extended typing and kinding rules are also straightforward, and are similar to those for the primitive types in $\lambda_{\texttt{OADT}\blacksquare}$. Figure 4.9 gives the rules for integer retraction (TI-Ret) and oblivious less-than (TI-OLe) as examples.

4.4 $\lambda_{\texttt{OADT}\blacksquare}$ in Action

To demonstrate the expressiveness of $\lambda_{\texttt{OADT}\blacksquare}$, 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 [48].

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 3.1. 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 3.1, 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 $\lambda_{\text{OADT}\textbf{+}}$. The following code snippet for an oblivious map function follows the recipe in Figure 4.1. Label annotations are omitted for brevity, and we use a boolean payload for simplicity.

```
fn ⌢map (f : 𝔹 → 𝔹) (k : ℕ) (t : ⌢tree k) : ⌢tree k =
  ⌢tree#s (map f (⌢tree#r k t)) k
```

The function argument of ⌢map takes a public boolean to public boolean, but ⌢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 ⌢map function could also be adapted any oblivious tree definition by simply replacing ⌢tree, ⌢tree#s, and ⌢tree#r.

4.5 An Unsafe Reference Semantics

Reasoning about program behaviors directly under tape semantics is not ideal, as tape semantics is a nonstandard semantics: we cannot directly apply our experience, theoretical frameworks or practical tools developed for standard functional languages. This section shows that most of the reasoning principles can be recovered by connecting tape semantics to a more standard, reference semantics, called *reveal semantics*.

Reveal semantics is an *unsafe*, big-step operational semantics for $\lambda_{\text{OADT}\textbf{+}}$, whose rules are presented in Figure 4.10. Its judgment $\Sigma \vdash \texttt{e} \Downarrow \texttt{v}$ evaluates an expression `e` to a value `v`, under a global context Σ which is elided for brevity in these rules. As $\lambda_{\text{OADT}\textbf{+}}$ is dependently typed, this relation also evaluates a type to an oblivious type value.

Most rules are straightforward, similar to the big-step semantics of other systems. However, unlike other languages, in E-Val a value `v` in reveal semantics is evaluated to the erasure of `v`, $\lfloor \texttt{v} \rfloor$, instead of itself. This erasure operation reduces all "dummy" leaky conditionals, i.e., those that are already weak values, even under the binder of a lambda abstraction. Its definition is simple: $\lfloor$⌢if $\texttt{e}_0$ then $\texttt{e}_1$ else $\texttt{e}_2\rfloor = \texttt{ite(b,}\lfloor\texttt{e}_1\rfloor\texttt{,}\lfloor\texttt{e}_2\rfloor\texttt{)}$ if $\lfloor \texttt{e}_0 \rfloor = \texttt{[b]}$ and both $\lfloor \texttt{e}_1 \rfloor$ and $\lfloor \texttt{e}_2 \rfloor$ are weak values, or otherwise the erasure is ⌢if $\lfloor \texttt{e}_0 \rfloor$ then $\lfloor \texttt{e}_1 \rfloor$ else $\lfloor \texttt{e}_2 \rfloor$. Other expressions simply erase their subterms recursively. This erasure operation is mainly used for establishing the equivalence to tape semantics. To see why this is necessary,

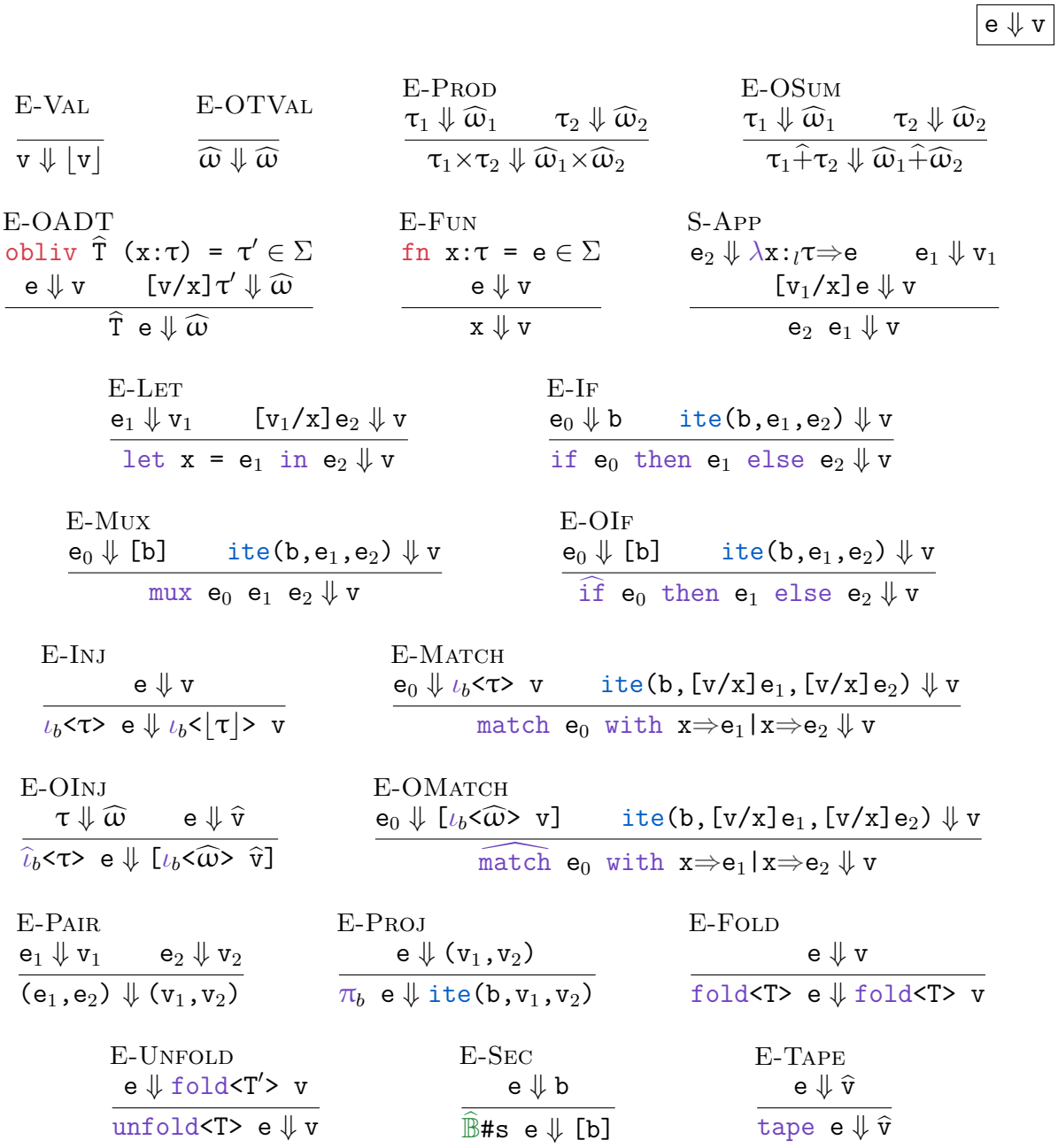


Figure 4.10. $\lambda_{\texttt{OADT+}}$ reveal semantics

consider the expression $\widehat{\texttt{if}}$ `[true]` `then` `1` `else` `2`. This expression evaluates to `1` under reveal semantics, but it is a stuck term in tape semantics to avoid revealing its private condition `[true]`. To connect reveal semantics to tape semantics, we have to consider a weaker "sameness" up to erasure. We further bake erasure into reveal semantics, so that an expression always evaluates to a sort of "canonical" value free of dummy leaky conditionals. E-Inj also uses this operation to erase a sum injection's type annotation, similar to E-Val.

E-OIf executes an $\widehat{\texttt{if}}$ similarly to the standard semantics of `if`, which allows us to reason about $\widehat{\texttt{if}}$ easily by treating it as a normal conditional. This behavior of course reveals the private conditions (in addition to the erasure), so reveal semantics is an unsafe semantics which should not be used for secure computation. However, it is suitable for executing retraction functions in order to reveal the computed secure output to the privileged parties. Note that reveal semantics does not guarantee to have the same termination behavior as tape semantics; it is entirely possible that a program terminates in reveal semantics but does not terminate in tape semantics, as tape semantics executes both branches of an $\widehat{\texttt{if}}$ regardless of its private condition. Readers may wonder if equi-termination can be achieved by forcing $\widehat{\texttt{if}}$ to similarly evaluate both branches in E-If. Unfortunately, $\widehat{\texttt{if}}$ can still indirectly do more computations under tape semantics. To see how, consider a program that has anonymous functions as branches: (`(`$\widehat{\texttt{if}}$ `[true]` `then` `(`λ`_`$\Rightarrow$`1)` `else` `(`λ`_`$\Rightarrow$`loop))` `()`. Under reveal semantics, the $\widehat{\texttt{if}}$ expression evaluates to λ`_`$\Rightarrow$`1`, and then the whole program terminates at `1`, even though both branches are evaluated. On the other hand, the function application is distributed into both branches under tape semantics, and the program reduces to $\widehat{\texttt{if}}$ `[true]` `then` `1` `else` `loop` which diverges due to the second branch. We made a similar design decision for `mux` in E-Mux, although it is also reasonable to make this rule closer to tape semantics.

4.5.1 Metatheory of Reveal Semantics

Reveal semantics enjoys several metatheoretic properties. First, it is deterministic.

Theorem 4.5.1 (Determinism of reveal semantics)**.** *If* $\texttt{e} \Downarrow \texttt{v}_1$ *and* $\texttt{e} \Downarrow \texttt{v}_2$*, then* $\texttt{v}_1 = \texttt{v}_2$*.*

Reveal semantics also guarantees type preservation with respect to $\lambda_{\texttt{OADT}\blacksquare}$'s type system.

Theorem 4.5.2 (Preservation of reveal semantics)**.** *If* $\Gamma \vdash \mathtt{e} :_l \tau$ *and* $\mathtt{e} \Downarrow \mathtt{v}$*, then* $\Gamma \vdash \mathtt{v} :_l \tau$*.*

Furthermore, the evaluated result $\mathtt{v}$ must be a value if the typing context Γ is empty.

A direct proof of Theorem 4.5.2 is tricky. We instead reduce the proof of preservation of reveal semantics to that of parallel reduction, by first showing parallel reduction refines reveal semantics.

Lemma 4.5.3 (Parallel reduction refines reveal semantics)**.** *If* $\mathtt{e} \Downarrow \mathtt{v}$*, then* $\mathtt{e} \Rrightarrow^* \mathtt{v}$*.*

Most importantly, we are able to relate reveal semantics to tape semantics using the following simulation theorem.

Theorem 4.5.4 (Simulation of reveal semantics)**.** *If* $\mathtt{e} \longrightarrow^* \mathtt{v}$*, then* $\mathtt{e} \Downarrow \lfloor \mathtt{v} \rfloor$*.*

If $\tau \longrightarrow^* \widehat{\omega}$*, then* $\tau \Downarrow \widehat{\omega}$*.*

Theorem 4.5.4 only requires $\mathtt{v}$ to be a weak value. This theorem relies on a crucial lemma that says the equivalence up to erasure preserves the reveal semantics relation, as follows.

Lemma 4.5.5. *If* $\mathtt{e}_1 \Downarrow \mathtt{v}$ *and* $\lfloor \mathtt{e}_1 \rfloor = \lfloor \mathtt{e}_2 \rfloor$*, then* $\mathtt{e}_2 \Downarrow \mathtt{v}$*.*

Note that Theorem 4.5.4 only considers one direction: if $\mathtt{e}$ evaluates to a value under reveal semantics, $\mathtt{e}$ does not necessarily reduce to any equivalent values under tape semantics, due to different termination behaviors exhibited by these two semantics. Nonetheless, if an expression terminates under both semantics, the results are equivalent.

Corollary 4.5.6. *If* $\mathtt{e} \longrightarrow^* \mathtt{v}$ *and* $\mathtt{e} \Downarrow \mathtt{v}'$*, then* $\lfloor \mathtt{v} \rfloor = \mathtt{v}'$*.*

If $\tau \longrightarrow^* \widehat{\omega}$ *and* $\tau \Downarrow \widehat{\omega}'$*, then* $\widehat{\omega} = \widehat{\omega}'$*.*

As a bonus, the determinism of reveal semantics allows us to prove that tape semantics is also deterministic (up to erasure) if the computation terminates (at some weak values).

Corollary 4.5.7 (Weak determinism of tape semantics)**.** *If* $\mathtt{e} \longrightarrow^* \mathtt{v}_1$ *and* $\mathtt{e} \longrightarrow^* \mathtt{v}_2$*, then* $\lfloor \mathtt{v}_1 \rfloor = \lfloor \mathtt{v}_2 \rfloor$*.*

The formalization of reveal semantics and all the proofs in this section are mechanized in the Coq proof assistant.

4.6 Deriving Secure Implementations

Following the recipe in Figure 4.1, one can easily implement a secure version of a public program written in a standard way. This strategy of deriving secure implementations can be automated, further levitating the user burden. This section develops an algorithm, `chase`, that converts a given public program to an equivalent secure version, by chasing the commuting diagrams as in Figure 4.1.

The algorithm takes as input a public program `e` (source program), usually a function, its type τ (source type), and a target type $\dot{\tau}$ which also serves as the security specification. It then outputs a secure program $\dot{e}$ of type $\dot{\tau}$. As an example, given the source program `lookup` of source type $\mathbb{Z} \to$ `tree` $\to \mathbb{B}$ (with $\top$ labels in all arguments elided) from Figure 3.1 and the target type Π`x:`$\mathbb{N}$`,` $\hat{\mathbb{Z}} \to \widehat{\texttt{tree}}$ `k` $\to \hat{\mathbb{B}}$ (with $\bot$ labels in all arguments elided), `chase(e;`τ`;`$\dot{\tau}$`)` generates the oblivious function $\widehat{\texttt{lookup}}$ in Figure 4.3. For simplicity, we require that the target type be in *prenex normal form*: $\dot{\tau}$ has a prefix of public views (e.g., Π`x:`$\mathbb{N}$`,` ...), followed by a non-dependent (i.e., quantifier-free) component (e.g., $\hat{\mathbb{Z}} \to \widehat{\texttt{tree}}$ `k` $\to \hat{\mathbb{B}}$). We also assume the source type τ is non-dependent, as the source program is supposed to be implemented in the standard fragment of $\lambda_{\text{OADT}\blacksquare}$. The labels are elided in the rest of this section for brevity, unless they are unclear from the context: a source type always has $\top$ labels for all of its arguments, while a targe type has $\bot$ labels.

Before we present the algorithm formally, we explore a few examples to demonstrate some design decisions of the algorithm. The first example converts a tree insertion function.

Example 4.6.1 (insert)**.**

```
chase(insert; ℤ → tree → tree; Πx:ℕ, ℤ̂ → tree̲ k → tree̲ (k+1)) ≡
  λk x t ⇒ tree̲#s (k+1) (insert (ℤ̂#r x) (tree̲#r k t))
```

As the result is an OADT, `chase` uses $\widehat{\texttt{tree}}$`#s` to convert the output of `insert` back to an oblivious tree. The `tape` keyword is not necessary in this example, because $\widehat{\texttt{tree}}$`#s` should have label $\bot$, implicitly having `tape`d the result in its definition. It is crucial that the programmers provide the correct public views in the target type signatures: if the return type in this example was $\widehat{\texttt{tree}}$ `k` instead, the resulting oblivious tree could get truncated

due to possible increase in the tree depth. Our algorithm assumes the provided public views are always "big" enough to hold the data.

The next example illustrates that our algorithm is able to handle polynomial types.

Example 4.6.2 (add1)**.**

```
chase(add1; ℤ + 𝟙 → ℤ + 𝟙; ℤ̂ +̂ 𝟙 → ℤ̂ +̂ 𝟙) ≡
  λn ⇒ let m = match n with
                | inl x ⇒ inl (ℤ̂#r x)
                | inr x ⇒ inr x
       in
       tape
         (match add1 m with
          | inl x ⇒ inl (ℤ̂#s x)
          | inr x ⇒ inr x)
```

The function `add1` adds 1 to the left injection of an option type $\mathbb{Z} + \mathbb{1}$ (i.e., an fmap). The generated program converts between sum type ($\mathbb{Z} + \mathbb{1}$) and oblivious sum type ($\hat{\mathbb{Z}} \hat{+} \mathbb{1}$), similar to the previous examples. However, the conversions are derived from the polynomial types (e.g., sum types), instead of provided by the users as section and retraction functions.

The last example is a standard higher-order function `map`.

Example 4.6.3 (map)**.**

```
chase(map; (ℤ→ℤ) → tree → tree; Πk:ℕ, (ℤ→ℤ) → tree̲ k → tree̲ k) ≡
  λk f t ⇒ tree̲#s k (map f (tree̲#r k t))
```

While this secure implementation is fairly easy to derive by chasing the commuting diagram, the higher-order argument ($\mathbb{Z}\to\mathbb{Z}$) in the target type is required to remain public. Converting higher-order arguments is possible for the primitive types in this example by "transporting" these arguments properly. However, it becomes challenging when the higher-order arguments use OADTs, which may force us to infer the public views of these OADTs. Thus, our algorithm opts to not convert higher-order arguments. This limitation is nonetheless insignificant in practice, because it is easier for a client of the generated secure `map` function to supply a public higher-order argument anyway, which also avoids unnecessary conversions.

$\boxed{\texttt{gsec}(\texttt{e};\tau;\dot{\tau}) = \dot{\texttt{e}}}$

GS-Unit
$$\frac{}{\texttt{gsec}(\texttt{e};\mathbb{1};\mathbb{1}) = \texttt{tape e}}$$

GS-Bool
$$\frac{}{\texttt{gsec}(\texttt{e};\mathbb{B};\widehat{\mathbb{B}}) = \texttt{tape } (\widehat{\mathbb{B}}\texttt{\#s e})}$$

GS-OADT
$$\frac{}{\texttt{gsec}(\texttt{e};\texttt{T};\widehat{\texttt{T}}\ \texttt{k}) = \widehat{\texttt{T}}\texttt{\#s k e}}$$

GS-Prod
$$\frac{\texttt{gsec}(\pi_1\ \texttt{e};\tau_1;\dot{\tau}_1) = \dot{\texttt{e}}_1 \qquad \texttt{gsec}(\pi_2\ \texttt{e};\tau_2;\dot{\tau}_2) = \dot{\texttt{e}}_2}{\texttt{gsec}(\texttt{e};\tau_1\times\tau_2;\dot{\tau}_1\times\dot{\tau}_2) = (\dot{\texttt{e}}_1,\dot{\texttt{e}}_2)}$$

GS-Sum
$$\frac{\texttt{gsec}(\texttt{x};\tau_1;\dot{\tau}_1) = \dot{\texttt{e}}_1 \qquad \texttt{gsec}(\texttt{x};\tau_2;\dot{\tau}_2) = \dot{\texttt{e}}_2}{\texttt{gsec}(\texttt{e};\tau_1+\tau_2;\dot{\tau}_1\mathbin{\widehat{+}}\dot{\tau}_2) = \texttt{tape } (\texttt{match e with x}\Rightarrow\widehat{\texttt{inl}}\ \dot{\texttt{e}}_1\texttt{|x}\Rightarrow\widehat{\texttt{inr}}\ \dot{\texttt{e}}_2)}$$

$\boxed{\texttt{gret}(\texttt{e};\tau;\dot{\tau}) = \dot{\texttt{e}}}$

GR-Id
$$\frac{}{\texttt{gret}(\texttt{e};\tau;\tau) = \texttt{e}}$$

GR-Bool
$$\frac{}{\texttt{gret}(\texttt{e};\widehat{\mathbb{B}};\mathbb{B}) = \widehat{\texttt{if}}\ \texttt{e then true else false}}$$

GR-OADT
$$\frac{}{\texttt{gret}(\texttt{e};\widehat{\texttt{T}}\ \texttt{k};\texttt{T}) = \widehat{\texttt{T}}\texttt{\#r k e}}$$

GR-Prod
$$\frac{\texttt{gret}(\pi_1\ \texttt{e};\tau_1;\dot{\tau}_1) = \dot{\texttt{e}}_1 \qquad \texttt{gret}(\pi_2\ \texttt{e};\tau_2;\dot{\tau}_2) = \dot{\texttt{e}}_2}{\texttt{gret}(\texttt{e};\tau_1\times\tau_2;\dot{\tau}_1\times\dot{\tau}_2) = (\dot{\texttt{e}}_1,\dot{\texttt{e}}_2)}$$

GR-Sum
$$\frac{\texttt{gret}(\texttt{x};\tau_1;\dot{\tau}_1) = \dot{\texttt{e}}_1 \qquad \texttt{gret}(\texttt{x};\tau_2;\dot{\tau}_2) = \dot{\texttt{e}}_2}{\texttt{gret}(\texttt{e};\tau_1+\tau_2;\dot{\tau}_1+\dot{\tau}_2) = \texttt{match e with x}\Rightarrow\texttt{inl}\ \dot{\texttt{e}}_1\texttt{|x}\Rightarrow\texttt{inr}\ \dot{\texttt{e}}_2}$$

GR-OSum
$$\frac{\texttt{gret}(\texttt{x};\tau_1;\dot{\tau}_1) = \dot{\texttt{e}}_1 \qquad \texttt{gret}(\texttt{x};\tau_2;\dot{\tau}_2) = \dot{\texttt{e}}_2}{\texttt{gret}(\texttt{e};\tau_1\mathbin{\widehat{+}}\tau_2;\dot{\tau}_1+\dot{\tau}_2) = \widehat{\texttt{match}}\ \texttt{e with x}\Rightarrow\texttt{inl}\ \dot{\texttt{e}}_1\texttt{|x}\Rightarrow\texttt{inr}\ \dot{\texttt{e}}_2}$$

Figure 4.11. Generalized section and retraction

Chapter 6 will describe a more general static approach to convert a public program to an equivalent secure program, which has better support for higher-order functions.

4.6.1 Derivation Algorithm

We first define the procedures that convert between public and oblivious polynomial types. The *generalized section* and *generalized retraction* procedures, $\texttt{gsec}(\texttt{e};\tau;\dot{\tau})$ and $\texttt{gret}(\texttt{e};\tau;\dot{\tau})$ respectively, generate a section computation or retraction computation that converts e from

$\boxed{\texttt{chase}(\texttt{e};\tau;\dot{\tau}) = \dot{\texttt{e}}}$

$$\frac{\overline{\texttt{x}:\tau'} \text{ is the public view prefix} \qquad \texttt{chase}(\texttt{e};\tau;\dot{\tau}) = \dot{\texttt{e}}}{\texttt{chase}(\texttt{e};\tau;\Pi\overline{\texttt{x}:\tau'},\dot{\tau}) = \lambda\overline{\texttt{x}:\tau'}\Rightarrow\dot{\texttt{e}}}\ \text{C-Prefix}$$

$$\frac{\texttt{gret}(\texttt{x};\dot{\tau}_1;\tau_1) = \texttt{r} \qquad \texttt{chase}(\texttt{e r};\tau_2;\dot{\tau}_2) = \dot{\texttt{e}}}{\texttt{chase}(\texttt{e};\tau_1\to\tau_2;\dot{\tau}_1\to\dot{\tau}_2) = \lambda\texttt{x}:\dot{\tau}_1\Rightarrow\dot{\texttt{e}}}\ \text{C-Ret}$$

$$\frac{\texttt{gsec}(\texttt{e};\tau;\dot{\tau}) = \dot{\texttt{e}}}{\texttt{chase}(\texttt{e};\tau;\dot{\tau}) = \dot{\texttt{e}}}\ \text{C-Sec}$$

Figure 4.12. Commuting diagram chasing

type τ to type $\dot{\tau}$. The result of `gsec` is always safe (i.e., with $\bot$ label), while that of `gret` may be leaky (i.e., with $\top$ label). On the other hand, the input `e` to `gret` is assumed safe. Figure 4.11 shows the definitions of these two procedures as inference rules. Note that these generators are partially defined; trying to convert a $\mathbb{1}$ to $\mathbb{B}$ will fail, for example. Most of these rules are straightforward. GS-OADT and GR-OADT outsource the conversions between ADTs and OADTs to the user-defined section and retraction functions, assuming $\hat{\texttt{T}}$ is an OADT of `T`. Generalized retraction also permits an identity conversion per GR-ID, if the source and target types are the same.

The main `chase` algorithm, presented in Figure 4.12 as inference rules, captures the key idea of the recipe from Figure 4.1. After introducing the public view arguments using C-Prefix, `chase` "decrypts" every function argument using the retraction computation generated by `gret` in C-Ret, and finally "encrypt" the result using the section computation generated by `gsec` in C-Sec. Similar to `gsec` and `gret`, `chase` is also partial: the algorithm fails if the source and target types do not match.

4.6.2 Metatheory of the Derivation Algorithm

The `chase` algorithm always generates well-typed and equivalent programs. The formalization of the algorithm and the proofs in this section are mechanized in the Coq proof assistant.

The well-typedness theorem, stated as follows, also guarantees that the generated programs are secure, thanks to Theorem 4.2.4.

Theorem 4.6.1 (Well-typedness of the derivation algorithm)**.** *Given a well-kinded target type* $\dot{\tau}$*, if* $\texttt{chase}(\texttt{e},\tau,\dot{\tau}) = \dot{\texttt{e}}$ *and* $\cdot \vdash \texttt{e} :_l \tau$*, then* $\cdot \vdash \dot{\texttt{e}} :_\bot \dot{\tau}$*.*

To establish the correctness of this algorithm, we consider two programs equivalent *up to revelation*, defined as follows in a logical relation style [30, 31].

Definition 4.6.1 (Equivalence of source and target programs)**.** The equivalence between a source program and a target program is defined as set-valued denotations indexed by their types: a value interpretation $\mathcal{V}[\![\tau;\dot{\tau}]\!]$ and an expression interpretation $\mathcal{E}[\![\tau;\dot{\tau}]\!]$. We say two values $\texttt{v}$ of type τ and $\dot{\texttt{v}}$ of type $\dot{\tau}$ are equivalent up to revelation if $(\texttt{v},\dot{\texttt{v}}) \in \mathcal{V}[\![\tau;\dot{\tau}]\!]$. The equivalence of two expressions are similarly defined using $\mathcal{E}[\![\tau;\dot{\tau}]\!]$.

$$\mathcal{V}[\![\tau;\tau]\!] = \{\ (\texttt{v},\texttt{v})\ \} \qquad \mathcal{V}[\![\mathbb{B};\widehat{\mathbb{B}}]\!] = \{\ (\texttt{b},\texttt{[b]})\ \} \qquad \mathcal{V}[\![\texttt{T};\widehat{\texttt{T}}\ \texttt{k}]\!] = \left\{\ (\texttt{v},\dot{\texttt{v}}) \;\middle|\; \widehat{\texttt{T}}\texttt{\#r}\ \texttt{k}\ \dot{\texttt{v}} \Downarrow \texttt{v}\ \right\}$$

$$\mathcal{V}[\![\tau_1{\times}\tau_2;\dot{\tau}_1{\times}\dot{\tau}_2]\!] = \{\ ((\texttt{v}_1,\texttt{v}_2),(\dot{\texttt{v}}_1,\dot{\texttt{v}}_2)) \mid (\texttt{v}_1,\dot{\texttt{v}}_1) \in \mathcal{V}[\![\tau_1;\dot{\tau}_1]\!] \wedge (\texttt{v}_2,\dot{\texttt{v}}_2) \in \mathcal{V}[\![\tau_2;\dot{\tau}_2]\!]\ \}$$

$$\mathcal{V}[\![\tau_1{+}\tau_2;\dot{\tau}_1{+}\dot{\tau}_2]\!] = \cup \begin{array}{l} \{\ (\texttt{inl}\ \texttt{v},\texttt{inl}\ \dot{\texttt{v}}) \mid (\texttt{v},\dot{\texttt{v}}) \in \mathcal{V}[\![\tau_1;\dot{\tau}_1]\!]\ \} \\ \{\ (\texttt{inr}\ \texttt{v},\texttt{inr}\ \dot{\texttt{v}}) \mid (\texttt{v},\dot{\texttt{v}}) \in \mathcal{V}[\![\tau_2;\dot{\tau}_2]\!]\ \} \end{array}$$

$$\mathcal{V}[\![\tau_1{+}\tau_2;\dot{\tau}_1\widehat{+}\dot{\tau}_2]\!] = \cup \begin{array}{l} \{\ (\texttt{inl}\ \texttt{v},\texttt{[inl}\ \dot{\texttt{v}}\texttt{]}) \mid (\texttt{v},\dot{\texttt{v}}) \in \mathcal{V}[\![\tau_1;\dot{\tau}_1]\!]\ \} \\ \{\ (\texttt{inr}\ \texttt{v},\texttt{[inr}\ \dot{\texttt{v}}\texttt{]}) \mid (\texttt{v},\dot{\texttt{v}}) \in \mathcal{V}[\![\tau_2;\dot{\tau}_2]\!]\ \} \end{array}$$

$$\mathcal{E}[\![\tau;\Pi\overline{\texttt{x}:\tau'},\dot{\tau}]\!] = \left\{\ (\texttt{e},\dot{\texttt{e}}) \;\middle|\; \begin{array}{r} \forall\overline{\texttt{v}}.\ (\forall i.\ \cdot \vdash \texttt{v}_i :_\bot \tau'_i) \implies \\ (\texttt{e},\dot{\texttt{e}}\ \overline{\texttt{v}}) \in \mathcal{E}[\![\tau;\dot{\tau}]\!] \end{array}\ \right\} \quad \text{if } \overline{\texttt{x}:\tau'} \text{ is the public view prefix}$$

$$\mathcal{E}[\![\tau_1{\to}\tau_2;\dot{\tau}_1{\to}\dot{\tau}_2]\!] = \left\{\ (\texttt{e},\dot{\texttt{e}}) \;\middle|\; \begin{array}{r} \forall\texttt{v}\ \dot{\texttt{v}}.\ \cdot \vdash \texttt{v} :_\bot \tau_1 \wedge \cdot \vdash \dot{\texttt{v}} :_\bot \dot{\tau}_1 \wedge (\texttt{v},\dot{\texttt{v}}) \in \mathcal{V}[\![\tau_1;\dot{\tau}_1]\!] \implies \\ (\texttt{e}\ \texttt{v},\dot{\texttt{e}}\ \dot{\texttt{v}}) \in \mathcal{E}[\![\tau_2;\dot{\tau}_2]\!] \end{array}\ \right\}$$

$$\mathcal{E}[\![\tau;\dot{\tau}]\!] = \{\ (\texttt{e},\dot{\texttt{e}}) \mid \forall\texttt{v}\ \dot{\texttt{v}}.\ \texttt{e} \longrightarrow^* \texttt{v} \wedge \dot{\texttt{e}} \longrightarrow^* \dot{\texttt{v}} \implies (\lfloor\texttt{v}\rfloor,\lfloor\dot{\texttt{v}}\rfloor) \in \mathcal{V}[\![\tau;\dot{\tau}]\!]\ \}$$

The value denotation of OADT shows why this equivalence is up to revelation: an OADT value $\dot{\texttt{v}}$ is equivalent to a public value $\texttt{v}$ if $\dot{\texttt{v}}$ can be "decrypted" to $\texttt{v}$. Note that we use reveal semantics for this "decryption". Unlike most logical relations in the literatures, two function

values are equivalent only if they are equal, reflecting the design decision of not converting higher-order arguments. The first two cases of expression denotation consume function arguments (including the public view arguments), and the last case says two (non-functional) expressions are equivalent if they reduce to equivalent values after erasure. The erasure is necessary here because expressions may reduce to weak values under tape semantics, and we consider two weak values, e.g., leaky conditionals, equivalent if their erasures (i.e., revelation) are equivalent. The last case of expression denotation also has an elided side condition that assumes the correctness of the public views in the target type; we refer interested readers to the Coq development for the full details.

We are now ready to state the correctness theorem for our diagram chasing algorithm.

Theorem 4.6.2 (Correctness of the derivation algorithm)**.** *If* $\texttt{chase}(\mathsf{e};\tau;\dot{\tau}) = \dot{\mathsf{e}}$*, then* $(\mathsf{e}, \dot{\mathsf{e}}) \in \mathcal{E}[\![\tau; \dot{\tau}]\!]$.

The direct proof of this theorem involves a lot of tedious and complex reasoning about tape semantics. We instead reduce the expression denotation (specifically the last case in Definition 4.6.1) to the one that uses reveal semantics from the previous section, which greatly simplifies the proofs.

4.7 Conclusion

While $\lambda_{\texttt{OADT}}$ enables secure applications that use private data structures and complex privacy policies, the lack of modularity forces users to implement ad-hoc secure versions of their programs for each desired policy. To allow programmers to write a single function and easily build secure programs with different public views, we have developed $\lambda_{\texttt{OADT}\boldsymbol{+}}$. This language is equipped with a novel semantics that repairs potential leaks without compromising the security guarantees of $\lambda_{\texttt{OADT}}$. An obliviousness theorem analogous to the one for $\lambda_{\texttt{OADT}}$ is mechanically proved. Despite being non-standard, tape semantics is equivalent to a standard semantics in a way that enables familiar reasoning. We have also developed an algorithm that derives an equivalent secure program automatically from a given policy specification and functionality written in a conventional way.

5. TAYPE: A POLICY-AGNOSTIC OBLIVIOUS LANGUAGE

The calculus introduced in Chapter 4, $\lambda_{\texttt{OADT}\blacksquare}$, permits security concerns to be decoupled from the program logic of an oblivious computation. The first key component is a dependent security type system in which *oblivious algebraic data types* could be encoded. These types equip private data with a *public view*, and guarantee that every private value with the same view is indistinguishable, i.e., an attacker can learn nothing about private data other than what its public view entails. The second key component is a novel *tape semantics* that uses security information provided by the type system to dynamically *repair* any potential information leaks at runtime. These components allow the programmer to write a program as normal, and then combine it with the desired public view, relying on the tape semantics to patch any information leaks. Unfortunately, $\lambda_{\texttt{OADT}\blacksquare}$ lacked an accompanying implementation.

This chapter presents a programming language for writing oblivious computations, Taype, that implements both the oblivious algebraic data types and tape semantics proposed in $\lambda_{\texttt{OADT}\blacksquare}$. Taype is equipped with a bidirectional type checker that enforces correct use of secure operations, and automatically infers annotations that enable potential leaks to be repaired. Our implementation is realized in a compilation pipeline, shown in Figure 5.1, that translates a Taype program and privacy policy (in the form of a public view) to an OCaml implementation which, when linked with a cryptographic backend, can be used by client programs to securely compute functions over private data. The main challenge that this toolchain must overcome is how to securely implement these two language features in a standard functional programming language. To implement oblivious types, our key idea is to represent dependently typed oblivious data using an *oblivious array*, and the types themselves as sizes indexing into an array. To implement tape semantics, we equip each type, including function types, with a *leaky structure* that reifies potentially leaky operations into a distinguished data type and inserts repairs when values of this type are used.

To summarize, the contributions of this chapter are as follows:

- We implement a bidirectional type checker for an extension of $\lambda_{\texttt{OADT}\blacksquare}$. Given a source program, this checker outputs a fully-annotated version in a typed core language called core Taype.

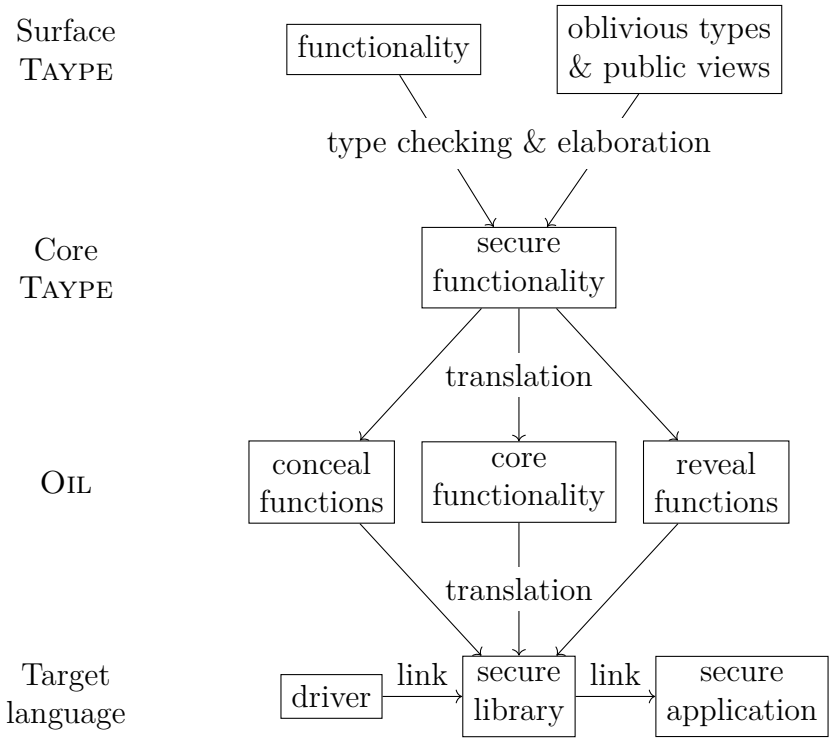


Figure 5.1. Compilation pipeline

- We present a translation from core Taype to Oil, an ML-style functional language with rank-1 polymorphism, built-in oblivious arrays, and secure array operations. In addition to translating the core functionality of the application, our translation also produces routines for concealing and revealing private data, which clients need to build a complete MPC application.

- We evaluate our implementation against several case studies and microbenchmarks. Our experiments feature a diverse set of computations and a range of security policies, including the medical record example in Chapter 1, and also demonstrate that tradeoffs between privacy and performance can be made easily with our approach.

An artifact containing the Coq mechanization of core Taype, the implementation of Taype, its source code, and the source for all the benchmarks in our experiments with instructions is publicly available [49].

5.1 Overview

To demonstrate our approach, consider a simple list membership predicate written in Taype, `elem`, shown in Figure 5.2. Suppose Alice, the owner of a list, and Bob, the owner of an integer, want to check if Bob's integer occurs in Alice's list, without revealing any information beyond their own input and the result. Similar to $\lambda_{\text{OADT}\blacksquare}$, Taype allows participants to choose what public information to share as part of the security policy, and its type system ensures that all public data and computation only depend on this public view. For example, Alice may be okay with sharing the size of the list, or with releasing some upper bound on its length.

```
data list = Nil | Cons ℤ list

fn elem : ℤ → list → 𝔹 =
  λy xs ⇒
    match xs with
    | Nil ⇒ false
    | Cons x xs′ ⇒
      if x = y then true
      else elem y xs′
```

Figure 5.2. List membership predicate

Using oblivious algebraic data types (Chapter 3) and tape semantics (Chapter 4), we can implement a secure version of this membership predicate by composing the standard implementation in Figure 5.2 and the desired privacy policy (i.e., OADTs and their section and retraction functions). Figure 5.3 shows the full implementation of an oblivious list $\widehat{\texttt{list}}$ and its the section and retraction functions, using the maximum length of the list as its public view.

While Chapter 4 formalized a core calculus of oblivious algebraic data types and tape semantics, $\lambda_{\text{OADT}\blacksquare}$, it lacked both an algorithmic type checker and implementation; this chapter presents the design and implementation of a language for oblivious computation with both.

5.1.1 Type Checking and Core Taype

The input to our compiler is a program written in *surface* Taype, such as `elem`, $\widehat{\texttt{list}}$`#r`, and $\widehat{\texttt{list}}$`#s` from Figure 5.2 and Figure 5.3. This language is equipped with a *bidirectional type checker* [50] that enforces correct use of secure and leaky operations, which ensures that

```
obliv ^list (k : ℤ) =
  if k = 0 then 𝟙
  else 𝟙 ^+ ℤ ^× ^list (k−1)

#[section]
fn ^list#s : (k : ℤ) → list → ^list k = λk xs ⇒
  if k = 0 then ()
  else tape (match xs with
             | Nil ⇒ ^inl ()
             | Cons x xs′ ⇒
               ^inr [tape (ℤ#s x), ^list#s (k−1) xs′])

#[retraction]
fn ^list#r : (k : ℤ) → ^list k → list = λk ⇒
  if k = 0 then λ_ ⇒ Nil
  else λxs ⇒ ^match xs with
              | ^inl _ ⇒ Nil
              | ^inr [x, xs′] ⇒
                Cons (ℤ#r x) (^list#r (k−1) xs′)

#[safe]
fn ^elem : (k : ℤ) → ℤ → ^list k → 𝔹 = λk x xs ⇒
  tape (𝔹#s (elem (ℤ#r x) (^list#r k xs)))
```

Figure 5.3. An oblivious implementation of $\widehat{\texttt{elem}}$

all well-typed programs are oblivious. After type checking, programs in this language are elaborated into an intermediate language called *core* Taype (Section 5.2).

Core Taype programs are fully annotated with types and, crucially, *leakage labels* (Chapter 4). Leakage labels track whether an expression contains potential leaks, i.e., whether it contains any leaky operations: we say an expression is *leaky* (labelled $\top$) if so, and *safe* (labelled $\bot$) otherwise. For example, $\widehat{\texttt{if}}$ `x then 1 else 2` is obviously leaky, as executing it naively leaks the private condition, while `mux x [1] [2]` and `false` are safe. In contrast to $\lambda_{\text{OADT}\blacksquare}$, an addition in core Taype is its *promotion* operation $\uparrow$, which explicitly casts a

```
fn elem_c :⊤ ℤ⊤ → list⊤ → 𝔹 =
  λ(y :⊤ ℤ) (xs :⊤ list) ⇒
    match xs with
    | Nil ⇒ ↑false
    | Cons x xs′ ⇒
      if x = y then ↑true
      else elem_c y xs′
```

```
fn elem_o : ℤ̃ → list̃ → 𝔹̃ =
  λy xs ⇒
    match̃_list if̂_𝔹 xs
      (prom_𝔹 false)
      (λx xs′ ⇒
        ĩf if̂_𝔹 (x ≅̃ y) (prom_𝔹 true)
           (elem_o y xs′))
```

Figure 5.4. A fully annotated implementation of `elem` in core Taype (`elem`$_c$) and its translation in Oil (`elem`$_o$)

safe expression to a leaky one, to help with the translation: the ↑`false` on the fourth line of `elem`$_c$ in Figure 5.4 is treated as a leaky expression, for example. Note that none of the label annotations or promotion are required in the surface language: they are either inferred or automatically inserted during elaboration.

5.1.2 Translating to Oil

The next compilation phase translates programs in core Taype into Oil, the *OADT intermediate language* (Section 5.3). Oil is an ML-style functional language with rank-1 polymorphism, extended with an *oblivious array* and its operations. These oblivious "primitives" will eventually be implemented by a cryptographic backend in the target language (Section 5.4), and Oil is agnostic to the particular implementation. Oil is designed to be a common subset of most standard functional languages, so that translating Oil to a particular language, e.g., OCaml, is straightforward. The main challenge in this phase is expressing the unique features of Taype that do not appear in conventional languages, particularly oblivious types (i.e., dependent types) and its tape semantics. Many of the core ingredients of this translation can be seen in Figure 5.4, which gives the fully elaborated implementation of `elem` in core Taype and its corresponding Oil version.

```
data Ã = prom_A A | if̂_A A Ã Ã
data Z̃ = r_Z A | prom_Z Z | if̂_Z A Z̃ Z̃

fn Z̃#s : Z̃ → Ã = λñ ⇒
  match ñ with
  | r_Z n̂ ⇒ prom_A n̂
  | prom_Z n ⇒ prom_A (Ẑ#s n)
  | if̂_Z b̂ ñ_1 ñ_2 ⇒
    if̂_A b̂ (Z̃#s ñ_1) (Z̃#s ñ_2)
```

```
fn list̃ : Z → N = λk ⇒
  if k = 0 then 0
  else 1 + max 0 (1 + list̃ (k−1))

fn tape : Ã → A = λã ⇒
  match ã with
  | prom_A â ⇒ â
  | if̂_A b̂ ã_1 ã_2 ⇒
    mux b̂ (tape ã_1) (tape ã_2)
```

Figure 5.5. Selected leaky types and functions in Oil

Translating Oblivious Type Definitions

As Oil does not have type-level computation, the definition of the oblivious type `list̃` is translated into a function from the public view to its *size* $\mathbb{N}$, shown in Figure 5.5. As we will see shortly, the size of an oblivious type can be used to access secure data residing in an oblivious array.

Translating Oblivious Types and Operations

We represent every oblivious type as a single uniform type, the oblivious array $\mathcal{A}$. This array is essentially a secure "buffer" holding the private data. For example, `list̃ k` in the type signature of `list̃#s`, in Figure 5.3, is translated to this array type $\mathcal{A}$, regardless of the public view `k`. Even though oblivious types are all flat arrays in Oil, the rich typing information is not lost: we can still extract the needed private information by (securely) accessing the array using the sizes of oblivious types, such as the aforementioned `list̃`. Oblivious operations are translated into corresponding oblivious array operations. For example, an oblivious pair of private data is simply the concatenation of the two corresponding arrays, and destructing an oblivious pair amounts to taking a slice of the array using the two components' sizes. Section 5.3.2 describes the translation of other oblivious constructs, including injections into oblivious sums.

Translating Tape Semantics

Implementing the tape semantics is the main challenge in translating from Taype to Oil. Recall the three key ideas of the tape semantics. First, leaky operations, such as $\widehat{\texttt{if}}$, are themselves irreducible. Second, the surrounding context of $\widehat{\texttt{if}}$ is distributed into both branches. Third, the `tape` operation repairs potential leaks, by turning $\widehat{\texttt{if}}$ into `mux`.

To implement the first idea, we translate leaky types, e.g., $\mathbb{Z}_\top$, into a *leaky representation*, e.g., $\widetilde{\mathbb{Z}}$, a data type that explicitly represents expressions that may contain potential leaks. By convention, we use $\tilde{\cdot}$ as a visual cue for a leaky representation, its associated functions and variables. The leaky representations of oblivious arrays ($\widetilde{\mathcal{A}}$) and integers ($\widetilde{\mathbb{Z}}$) are shown in Figure 5.5. The only way to build a leaky oblivious data type is to promote a safe one or using a leaky conditional, so we simply encode these leaky operations as the constructors of its leaky representation $\widetilde{\mathcal{A}}$. $\widetilde{\mathbb{Z}}$ also includes both of these constructors, as well as its own retraction operation. This encoding trivially makes the leaky operations irreducible. Leaky representations of ADTs are built using a similar strategy (Section 5.3.2). Every leaky representation needs to have reified versions of `prom` and $\widehat{\texttt{if}}$, because $\uparrow$ and $\widehat{\texttt{if}}$ can be applied to any Taype type. During translation, they are instantiated using a process similar to typeclass resolution: the promotion of `false` in $\texttt{elem}_\texttt{c}$, for example, is resolved to $\texttt{prom}_\mathbb{B}$.

To distribute surrounding contexts into leaky constructs, we instrument the possible surrounding contexts to handle the leaky operations, by translating them into recursive functions following the tape semantics. For example, $\widehat{\mathbb{Z}}$`#s` is translated to $\widetilde{\mathbb{Z}}$`#s`, also shown in Figure 5.5. Observe the last case of $\widetilde{\mathbb{Z}}$`#s`, which has recursive calls to itself in both $\widehat{\texttt{if}}_\mathbb{Z}$ branches: this aligns with our intuition from the execution trace of the following example:

```
      tape (ℤ#s (if [true] then 3 else 4))
⟶     tape (if [true] then ℤ#s 3 else ℤ#s 4)
⟶*    tape (if [true] then [3] else [4]) ⟶ mux [true] [3] [4] ⟶ [3]
```

On the other hand, $\widetilde{\mathbb{Z}\texttt{\#s}}$'s handling of $\texttt{r}_{\mathbb{Z}}$ matches the following execution trace, for example:

$$
\begin{aligned}
& \texttt{tape}\ (\widehat{\mathbb{Z}}\texttt{\#s}\ (\widehat{\mathbb{Z}}\texttt{\#r}\ \texttt{[3]}\ +\ \widehat{\mathbb{Z}}\texttt{\#r}\ \texttt{[2]})) \\
\longrightarrow\ & \texttt{tape}\ (\widehat{\mathbb{Z}}\texttt{\#s}\ (\widehat{\mathbb{Z}}\texttt{\#r}\ (\texttt{[3]}\ \widehat{+}\ \texttt{[2]}))) \\
\longrightarrow\ & \texttt{tape}\ (\widehat{\mathbb{Z}}\texttt{\#s}\ (\widehat{\mathbb{Z}}\texttt{\#r}\ \texttt{[5]})) \longrightarrow \texttt{tape}\ \texttt{[5]} \longrightarrow \texttt{[5]}
\end{aligned}
$$

Our solution to patching leaky computation without `tape` is encapsulated in the definition of $\widetilde{\texttt{tape}}$, shown in Figure 5.5. The function simply converts all reified $\widehat{\texttt{if}}$s into `mux`s, and is essentially a transcription of `tape`'s evaluation rule.

To see how all these fit together, the initial expression from the previous example, $\texttt{tape}\ (\widehat{\mathbb{Z}}\texttt{\#s}\ (\widehat{\texttt{if}}\ \texttt{[true]}\ \texttt{then}\ \texttt{3}\ \texttt{else}\ \texttt{4}))$, is translated into the following Oil program:
$\widetilde{\texttt{tape}}\ (\widetilde{\mathbb{Z}\texttt{\#s}}\ (\widehat{\texttt{if}}_{\mathbb{Z}}\ \texttt{[true]}\ (\texttt{prom}_{\mathbb{Z}}\ \texttt{3})\ (\texttt{prom}_{\mathbb{Z}}\ \texttt{4})))$

Readers can verify that evaluating this program using a standard semantics produces the same behavior seen in the previous execution trace generated by tape semantics.

5.2 Taype, Formally

This section describes the fully annotated core Taype language. This language is inspired by the core calculus $\lambda_{\texttt{OADT}\blacksquare}$, but adds several features to aid its translation to Oil, including oblivious products, label promotion, ML-style ADT definitions, and explicit and uniform label checking. The user-facing version of Taype allows for many annotations to be omitted; these annotations are automatically inferred by our bidirectional type checker (Section 5.2.5) before translation to Oil (Section 5.3).

5.2.1 Syntax

Figure 5.6 shows the syntax of core Taype. Types and terms are in the same syntactic class, similar to $\lambda_{\texttt{OADT}\blacksquare}$. By convention, we use `e` for terms and τ for types whenever possible. A core Taype program consists of a global context of ADTs, functions and oblivious types, defined using `data`, `fn` and `obliv` respectively. We use lower case `x` for function and variable names, `C` for constructors, `T` for ADT names and $\widehat{\texttt{T}}$ for OADT names. Each constructor of an

e,τ	::=		EXPRESSIONS:
	|	$\mathbb{B} \mid \mathbb{Z} \mid \tau\times\tau$	standard types
	|	$\mathbb{1} \mid \widehat{\mathbb{B}} \mid \widehat{\mathbb{Z}} \mid \tau\mathbin{\widehat{\times}}\tau \mid \tau\mathbin{\widehat{+}}\tau$	oblivious types
	|	`() | b | n | x | T`	literals and variables
	|	Π`x:`$_l\tau$`,τ` $\mid$ λ`x:`$_l\tau$`⇒e`	dependent function
	|	`e`$\oplus$`e` $\mid$ `e`$\widehat{\oplus}$`e`	(oblivious) integer operations
	|	`e e | C e |` $\widehat{\texttt{T}}$ `e`	applications
	|	`let x:`$_l\tau$ `= e in e`	let binding
	|	`if`$_\tau$ `e then e else e`	conditional
	|	`mux e e e`	atomic conditional
	|	`(e,e) | [e,e]`	(oblivious) pair
	|	$\widehat{\iota}_b$`<τ> e`	oblivious sum injection
	|	`match`$_\tau$ `e with (x,x)⇒e`	product elimination
	|	$\widehat{\texttt{match}}$ `e:τ`$\widehat{\times}$`τ with [x,x]⇒e`	oblivious product elimination
	|	`match`$_\tau$ `e with` $\overline{\texttt{C x⇒e}}$	ADT elimination
	|	$\widehat{\texttt{match}}_\tau$ `e:τ`$\widehat{+}$`τ with x⇒e|x⇒e`	oblivious sum elimination
	|	$\widehat{\mathbb{B}}$`#s e |` $\widehat{\mathbb{Z}}$`#s e`	primitive sections
	|	$\widehat{\mathbb{Z}}$`#r e`	primitive integer retraction
	|	$\widehat{\texttt{if}}$ `e then e else e`	leaky conditional
	|	↑`e`	promotion
	|	`tape e`	tape operation
	|	`[b] | [n] |` [ι_b`<`$\widehat{\omega}$`> v̂`]	runtime boxed values
D	::=		GLOBAL DEFINITIONS:
	|	`data T =` $\overline{\texttt{C τ}}$	algebraic data type definition
	|	`fn x:`$_l\tau$ `= e`	(recursive) function definition
	|	`obliv` $\widehat{\texttt{T}}$ `(x:τ) = τ`	(recursive) oblivious type definition
l	::=	$\top \mid \bot$	LEAKAGE LABEL
$\widehat{\omega}$	::=	$\mathbb{1} \mid \mathbb{B} \mid \widehat{\mathbb{Z}} \mid \widehat{\omega}\mathbin{\widehat{\times}}\widehat{\omega} \mid \widehat{\omega}\mathbin{\widehat{+}}\widehat{\omega}$	OBLIVIOUS TYPE VALUES
$\hat{v}$	::=	`() | [b] | [n] | [v̂,v̂] |` [ι_b`<`$\widehat{\omega}$`> v̂`]	OBLIVIOUS VALUES
v	::=	$\widehat{\texttt{if}}$ `[b] then v else v` $\mid$ ↑`v` $\mid$ $\widehat{\mathbb{Z}}$`#r v`	WEAK VALUES
	|	`v̂ | b | n | (v,v) |` λ`x:`$_l\tau$`⇒e | C v`	VALUES

Figure 5.6. Core TAYPE syntax: the annotations marked in gray are either omitted (e.g., promotion, labels) or optional (e.g., argument types to dependent functions) in the user-facing surface language; the expressions marked in brown are restricted to be variables in administrative normal form.

ADT definition takes exactly one argument for simplicity, but this argument can be $\mathbb{1}$ for constructors that takes no argument, or a tuple of types for constructors that have multiple arguments..

Taype features a number of oblivious types and constructs, including oblivious integers, booleans, sums, and conditionals. The primitive section functions $\widehat{\mathbb{B}}$`#s` and $\widehat{\mathbb{Z}}$`#s` "encrypt" boolean and integer values respectively. Unlike $\lambda_{\texttt{OADT}\blacksquare}$, Taype also includes oblivious product types ($\widehat{\times}$), which are built using `[·,·]` and require both of their components to be oblivious and non-leaky. The atomic conditional `mux`, discussed in Chapter 3, fully evaluates both of its branches before taking an atomic step to its final result.

In core Taype, the arguments of dependent function types and lambda abstractions are annotated with a *leakage label* that indicates if they accept leaky inputs. We say that an Taype expression is *leaky* (i.e., has the label $\top$) if it contain potential leaks, e.g., uses some leaky operations, and say that it is *safe* otherwise. Standard conditional, product and ADT pattern matching expressions are annotated with the result type, while the elimination forms for oblivious products and sums are also annotated with the type of the discriminee, to help with their translation. All of these annotations are inferred by our type checker. For brevity, we omit them from now if they can be inferred from the context. Note that product elimination in Taype is defined (positively) using a pattern matching expression, instead of (negatively) using projection as in $\lambda_{\texttt{OADT}\blacksquare}$. Similarly, ADT introduction and elimination forms are defined in ML-style, using constructors and pattern matching, instead of `fold` and `unfold`.

Leaky conditionals, $\widehat{\texttt{match}}$ expressions, and `tape` operations play a key role in the semantics of Taype. The leaky conditional $\widehat{\texttt{if}}$ is similar to `mux`, but it allows its branches to be non-oblivious; $\widehat{\texttt{match}}$ analysis for oblivious sums is similar. The promotion operation $\uparrow$ explicitly converts a safe expression to a leaky one. Integer retraction $\widehat{\mathbb{Z}}$`#r` would reveal its oblivious argument if implemented naively (as would the other leaky operations), but this is disallowed by the semantics of Taype.

Oblivious type values ($\widehat{\omega}$), oblivious values ($\widehat{\mathtt{v}}$) and runtime boxed values like `[b]` are all identical to the definitions in $\lambda_{\texttt{OADT}\blacksquare}$. Weak values ($\mathtt{v}$) extends the one in $\lambda_{\texttt{OADT}\blacksquare}$ with promotion, primitive integer retraction and ML-style ADT values.

5.2.2 Semantics

Figure 5.7 shows a selection of the small-step operational semantics rules of core Taype (the full rules are in Appendix A.1). The judgment $\mathtt{e} \longrightarrow \mathtt{e}'$ means $\mathtt{e}$ steps to $\mathtt{e}'$ under a fixed global context of definitions, which we elide. S-Ctx takes a step in a subexpression according to an evaluation context $\mathcal{E}$, also given in Figure 5.7. Oblivious types are subject to reduction, as seen in the evaluation contexts involving $\widehat{\times}$, $\widehat{+}$ and $\widehat{\imath}$. To prevent information leaks, all subexpressions of a `mux` are fully evaluated by first applying the S-Ctx rule with the corresponding evaluation contexts, before `mux` itself can be reduced by the S-Mux rule. The semantics of $\widehat{\texttt{if}}$ is similar. Note that an $\widehat{\texttt{if}}$ expression is in normal form once all its components are normalized, in order to avoid revealing its private condition.

The evaluation rules involving `tape` are one of the distinguishing features of Taype. S-OIf captures the idea that the leaky conditional $\widehat{\texttt{if}}$, while in normal form, can still make progress by distributing its context into both branches. *Leaky contexts* $\widehat{\mathcal{E}}$ define what contexts can be distributed in this manner: other contexts are either ruled out by the type system or not useful. S-TapeOIf and S-TapeProm show how the `tape` operation repairs an expression with potential leaks. In addition to turning $\widehat{\texttt{if}}$ into `mux`, the enclosing `tape` is pushed inside the branches of `mux` in order to ensure any leaks they contain are also patched. On the other hand, S-TapeProm simply extracts the safe oblivious value from a promotion. In contrast to $\lambda_{\texttt{OADT}\blacksquare}$, Taype also includes new rules for promoted expressions. S-SecRetInt repairs the leaky operation $\widehat{\mathbb{Z}}\texttt{\#r}$ by canceling it with $\widehat{\mathbb{Z}}\texttt{\#s}$, for example, but it also promotes the resulting oblivious integer in order to preserve the leakage label. S-SecIntProm shows how promotion interacts with $\widehat{\mathbb{Z}}\texttt{\#s}$.

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

S-Ctx
$$\frac{\mathtt{e} \longrightarrow \mathtt{e}'}{\mathcal{E}[\mathtt{e}] \longrightarrow \mathcal{E}[\mathtt{e}']}$$

S-Mux
$$\frac{}{\texttt{mux [b] } \mathtt{v}_1 \ \mathtt{v}_2 \longrightarrow \texttt{ite(b,}\mathtt{v}_1\texttt{,}\mathtt{v}_2\texttt{)}}$$

S-SecRetInt
$$\frac{}{\hat{\mathbb{Z}}\texttt{\#s (}\hat{\mathbb{Z}}\texttt{\#r [n])} \longrightarrow \uparrow\texttt{[n]}}$$

S-SecIntProm
$$\frac{}{\hat{\mathbb{Z}}\texttt{\#s (}\uparrow\texttt{n)} \longrightarrow \uparrow\texttt{(}\hat{\mathbb{Z}}\texttt{\#s n)}}$$

S-OMatch
$$\frac{\hat{\mathtt{v}}_1 \Leftarrow \widehat{\omega}_1 \qquad \hat{\mathtt{v}}_2 \Leftarrow \widehat{\omega}_2}{\widehat{\texttt{match}}\ \texttt{[}\iota_b\texttt{<}\widehat{\omega}_1 \mathbin{\hat{+}} \widehat{\omega}_2\texttt{>}\ \hat{\mathtt{v}}\texttt{]}\ \texttt{with x}\Rightarrow\mathtt{e}_1\texttt{|x}\Rightarrow\mathtt{e}_2 \longrightarrow \begin{array}{l}\widehat{\texttt{if [b] then ite(b,[}\hat{\mathtt{v}}\texttt{/x]}\mathtt{e}_1\texttt{,[}\hat{\mathtt{v}}_1\texttt{/x]}\mathtt{e}_1\texttt{)}}\\ \texttt{else ite(b,[}\hat{\mathtt{v}}_2\texttt{/x]}\mathtt{e}_2\texttt{,[}\hat{\mathtt{v}}\texttt{/x]}\mathtt{e}_2\texttt{)}\end{array}}$$

S-OIf
$$\frac{}{\hat{\mathcal{E}}[\widehat{\texttt{if [b] then }}\mathtt{v}_1\texttt{ else }\mathtt{v}_2] \longrightarrow \widehat{\texttt{if [b] then }}\hat{\mathcal{E}}[\mathtt{v}_1]\texttt{ else }\hat{\mathcal{E}}[\mathtt{v}_2]}$$

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

S-TapeProm
$$\frac{}{\texttt{tape (}\uparrow\texttt{v)} \longrightarrow \texttt{v}}$$

Evaluation Contexts

$$\begin{array}{rcl}\mathcal{E} & ::= & \square\mathbin{\hat{\times}}\tau \mid \widehat{\omega}\mathbin{\hat{\times}}\square \mid \square\mathbin{\hat{+}}\tau \mid \widehat{\omega}\mathbin{\hat{+}}\square \\ & \mid & \texttt{mux } \square\ \texttt{e e} \mid \texttt{mux v } \square\ \texttt{e} \\ & \mid & \texttt{mux v v } \square \\ & \mid & \hat{\iota}_b\texttt{<}\square\texttt{> e} \mid \hat{\iota}_b\texttt{<}\widehat{\omega}\texttt{>}\ \square \\ & \mid & \widehat{\texttt{if }}\square\texttt{ then e else e} \\ & \mid & \widehat{\texttt{if v then }}\square\texttt{ else e} \\ & \mid & \widehat{\texttt{if v then v else }}\square \\ & \mid & \ldots\end{array}$$

Leaky Contexts

$$\begin{array}{rcl}\hat{\mathcal{E}} & ::= & \square\ \texttt{v} \\ & \mid & \texttt{if } \square\texttt{ then e else e} \\ & \mid & \texttt{match } \square\texttt{ with } \overline{\texttt{C x}\Rightarrow\texttt{e}} \\ & \mid & \texttt{match } \square\texttt{ with (x}_1\texttt{,x}_2\texttt{)}\Rightarrow\texttt{e} \\ & \mid & \hat{\mathbb{B}}\texttt{\#s } \square \mid \hat{\mathbb{Z}}\texttt{\#s } \square \\ & \mid & \square\mathbin{\hat{\oplus}}\texttt{v} \mid \texttt{v}\mathbin{\hat{\oplus}}\square\end{array}$$

Figure 5.7. Selected small-step semantics rules of core Taype

To see how these rules work, consider a core Taype version of the example from Section 5.1, which produces the following execution trace:

```
      tape (ℤ̂#s (if̂ [true] then ↑3 else ↑4))
⟶     tape (if̂ [true] then ℤ̂#s ↑3 else ℤ̂#s ↑4)
⟶     tape (if̂ [true] then ↑(ℤ̂#s 3) else ℤ̂#s ↑4)
⟶     tape (if̂ [true] then ↑[3] else ℤ̂#s ↑4)
⟶*    tape (if̂ [True] then ↑[3] else ↑[4])
⟶     mux [true] (tape ↑[3]) (tape ↑[4])
⟶*    mux [true] [3] [4] ⟶ [3]
```

The S-OMatch rule reduces a leaky case analysis of an oblivious sum to an $\widehat{\texttt{if}}$ using the discriminee's private tag. Similar to $\lambda_{\text{OADT}\blacksquare}$, the pattern variable `x` in the "correct" branch is of course instantiated with the injection payload, while the one in the "wrong" branch is instantiated with an arbitrary oblivious value of the right type. For example, if the discriminee of a leaky case is `[inl<`$\widehat{\mathbb{Z}}\widehat{+}\widehat{\mathbb{Z}}\widehat{\times}\widehat{\mathbb{Z}}$`> 1]`, the pattern variable in the second branch can be substituted by `[[0],[0]]`, `[[0],[1]]`, or any other oblivious pair of oblivious integers.

5.2.3 Type System

Figure 5.8 shows an illustrative subset of the typing rules of core Taype (the full set of typing and kinding rules are in Appendix A.2). The judgment $\Gamma \vdash \mathtt{e} :_l \tau$ types the expression `e` with type τ and leakage label l, under the typing context Γ (and an elided global typing context). Some typing rules refer to the kinding judgment $\Gamma \vdash \tau :: \kappa$, which also classifies the security of a type; oblivious types have the kind $*^{\mathsf{O}}$, for example.

Taype features a security-type system [27] that ensures well-typed programs protect their private data. To do so, this type system enforces a few key policies. First, oblivious types can only be built from oblivious types, which is enforced by the kinding rules. Otherwise, an attacker can infer the private tag of an oblivious sum, such as $\mathbb{B}\widehat{+}\mathbb{Z}$, by observing the payload. Oblivious products have the same requirement, although this is mainly to aid in

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

$$\textsc{T-Conv}\ \frac{\Gamma \vdash \mathtt{e} :_l \tau \qquad \tau \equiv \tau' \qquad \Gamma \vdash \tau' :: *}{\Gamma \vdash \mathtt{e} :_l \tau'} \qquad \textsc{T-Abs}\ \frac{\mathtt{x} :_{l_1} \tau_1, \Gamma \vdash \mathtt{e} :_{l_2} \tau_2 \qquad \Gamma \vdash \tau_1 :: *}{\Gamma \vdash \lambda \mathtt{x}:_{l_1}\tau_1 \Rightarrow \mathtt{e} :_{l_2} \Pi \mathtt{x}:_{l_1}\tau_1, \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash \mathtt{e}_2 :_{l_2} \Pi \mathtt{x}:_{l_1}\tau_1, \tau_2 \qquad \Gamma \vdash \mathtt{e}_1 :_{l_1} \tau_1}{\Gamma \vdash \mathtt{e}_2\ \mathtt{e}_1 :_{l_2} [\mathtt{e}_1/\mathtt{x}]\tau_2}$$

$$\textsc{T-Pair}\ \frac{\Gamma \vdash \mathtt{e}_1 :_l \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :_l \tau_2}{\Gamma \vdash (\mathtt{e}_1, \mathtt{e}_2) :_l \tau_1 \times \tau_2} \qquad \textsc{T-PMatchNoDep}\ \frac{\Gamma \vdash \mathtt{e}_0 :_{l_0} \tau_1 \times \tau_2 \qquad l_0 \sqsubseteq l \qquad \mathtt{x}_1 :_{l_0} \tau_1, \mathtt{x}_2 :_{l_0} \tau_2, \Gamma \vdash \mathtt{e} :_l \tau}{\Gamma \vdash \mathtt{match}_\tau\ \mathtt{e}_0\ \mathtt{with}\ (\mathtt{x}_1, \mathtt{x}_2) \Rightarrow \mathtt{e} :_l \tau}$$

$$\textsc{T-IfNoDep}\ \frac{\Gamma \vdash \mathtt{e}_0 :_{l_0} \mathbb{B} \qquad l_0 \sqsubseteq l \qquad \Gamma \vdash \mathtt{e}_1 :_l \tau \qquad \Gamma \vdash \mathtt{e}_2 :_l \tau}{\Gamma \vdash \mathtt{if}_\tau\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 :_l \tau} \qquad \textsc{T-Mux}\ \frac{\Gamma \vdash \mathtt{e}_0 :_\bot \widehat{\mathbb{B}} \qquad \Gamma \vdash \tau :: *^0 \qquad \Gamma \vdash \mathtt{e}_1 :_\bot \tau \qquad \Gamma \vdash \mathtt{e}_2 :_\bot \tau}{\Gamma \vdash \mathtt{mux}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2 :_\bot \tau}$$

$$\textsc{T-OPair}\ \frac{\Gamma \vdash \mathtt{e}_1 :_\bot \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :_\bot \tau_2 \qquad \Gamma \vdash \tau_1 :: *^0 \qquad \Gamma \vdash \tau_2 :: *^0}{\Gamma \vdash [\mathtt{e}_1, \mathtt{e}_2] :_\bot \tau_1 \mathbin{\widehat{\times}} \tau_2} \qquad \textsc{T-SecInt}\ \frac{\Gamma \vdash \mathtt{e} :_l \mathbb{Z}}{\Gamma \vdash \widehat{\mathbb{Z}}\mathtt{\#s}\ \mathtt{e} :_l \widehat{\mathbb{Z}}} \qquad \textsc{T-RetInt}\ \frac{\Gamma \vdash \mathtt{e} :_\bot \widehat{\mathbb{Z}}}{\Gamma \vdash \widehat{\mathbb{Z}}\mathtt{\#r}\ \mathtt{e} :_\top \mathbb{Z}}$$

$$\textsc{T-OIf}\ \frac{\Gamma \vdash \mathtt{e}_0 :_\bot \widehat{\mathbb{B}} \qquad \Gamma \vdash \mathtt{e}_1 :_\top \tau \qquad \Gamma \vdash \mathtt{e}_2 :_\top \tau}{\Gamma \vdash \widehat{\mathtt{if}}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 :_\top \tau} \qquad \textsc{T-Promote}\ \frac{\Gamma \vdash \mathtt{e} :_\bot \tau}{\Gamma \vdash \uparrow \mathtt{e} :_\top \tau} \qquad \textsc{T-Tape}\ \frac{\Gamma \vdash \mathtt{e} :_\top \tau \qquad \Gamma \vdash \tau :: *^0}{\Gamma \vdash \mathtt{tape}\ \mathtt{e} :_\bot \tau}$$

Figure 5.8. Selected core TAYPE typing rules

translation. Second, oblivious control flow constructs like `mux` can only be applied to oblivious terms, otherwise their public result could reveal information about their condition. As an example, `mux [b] 1 2` is ill-typed, because an attacker can learn the value of `b` by observing its result. This policy is enforced by the kinding assumptions of the form $\Gamma \vdash \tau :: *^0$ in T-MUX and T-OPAIR. Third, types are not allowed to depend on leaky terms. The type `if` $\widehat{\mathbb{Z}}$`#r [0] = 0 then` $\mathbb{1}$ `else` $\widehat{\mathbb{B}}$ is not valid, for example, since the leaks in the condition can not be repaired. Thus, we require that any terms appearing in types to be labeled as non-leaky ($\bot$). Fourth, the argument to `tape` must be oblivious (T-TAPE). This ensures that leaky terms will *eventually* reduce to an oblivious value or a $\widehat{\mathtt{if}}$ tree of oblivious values that can then be repaired by, e.g., S-TAPEPROM or S-TAPEOIF. Intuitively, the $\bot$ label in

the conclusion of T-Tape signifies that the taped expression can be treated as non-leaky by its surrounding computation, as all leaks have been "patched up". Finally, all oblivious components in the typing rules have the $\bot$ label. All the labels in T-Mux and T-OPair are $\bot$, and the oblivious condition of $\widehat{\texttt{if}}$ (T-OIf) is also safe, for example. While this requirement is not crucial for security, it simplifies the type system and aids in our translation to Oil. Note that we can always apply `tape` to leaky oblivious expressions to make them safe, so this design does not harm the expressivity of well-typed Taype programs.

In addition to the above policies inherited from $\lambda_{\texttt{OADT}\blacksquare}$, Taype's type system imposes three more requirements that help our translation. First, safe terms must be explicitly converted to leaky ones using $\uparrow$. Thus, T-Conv requires convertible expressions to have the same label. Second, we usually require subexpressions to have the same label: the two components in T-Pair have the same label l in Taype, for example. T-IfNoDep similarly requires both branches to have the same label. Its condition, however, is permitted to have a lower label. A similar requirement is particularly important for the case analysis of products and ADTs: each branch needs to use its pattern variables in a manner that is at least as safe as the discriminee. Third, we require all *possibly* leaky subexpressions to be labelled as leaky. The branches in T-OIf and the argument to T-Tape have label $\top$, even though they can technically also be typed at $\bot$: applying these rules to an expression with a safe subterm requires explicit promotion. Note that programmers do not need to do these explicit label conversion in the surface language, as $\uparrow$ is automatically inserted by the typing algorithm presented in Section 5.2.5.

5.2.4 Type Safety and Obliviousness

Given a well-typed global context, core Taype enjoys standard progress and preservation properties. Its type system also provides a strong security guarantee similar to $\lambda_{\texttt{OADT}\blacksquare}$: an adversary cannot infer any private information from a well-typed core Taype program, even when they can observe each of its execution steps.

Theorem 5.2.1 (Obliviousness). *If* $\mathsf{e}_1 \approx \mathsf{e}_2$ *and* $\cdot \vdash \mathsf{e}_1 :_{l_1} \tau_1$ *and* $\cdot \vdash \mathsf{e}_2 :_{l_2} \tau_2$*, then*

1. $\mathsf{e}_1 \longrightarrow^n \mathsf{e}_1'$ *if and only if* $\mathsf{e}_2 \longrightarrow^n \mathsf{e}_2'$ *for some* e_2'.

2. if $\mathsf{e}_1 \longrightarrow^n \mathsf{e}_1'$ *and* $\mathsf{e}_2 \longrightarrow^n \mathsf{e}_2'$*, then* $\mathsf{e}_1' \approx \mathsf{e}_2'$*.*

Here, $\mathsf{e} \approx \mathsf{e}'$ means the two expressions are *indistinguishable*, i.e., they only differ in their oblivious values, and $\mathsf{e} \longrightarrow^n \mathsf{e}'$ means e reduces to e' in exactly n steps. Intuitively, the obliviousness theorem says that a pair of well-typed core Taype programs that are indistinguishable produce traces that are pairwise indistinguishable.

We have formalized a version of core Taype in Coq, including proofs of soundness and obliviousness for the calculus, based on the mechanization from Chapter 4. In contrast to that development, this calculus includes the new features of Taype: oblivious products, label promotion and explicit and uniform label checking.

5.2.5 Surface Language and Bidirectional Type Checker

The source language of our compiler is a more user friendly version of core Taype. This language allows type annotations to be omitted, and does not require label annotations or explicit promotion operations. Its syntax is effectively that of Figure 5.6 with the gray annotations removed and with an additional type ascription $\texttt{(e:}\tau\texttt{)}$. Our type checker elaborates programs in this surface language into fully annotated core Taype programs in ANF [51]. Our inference algorithm is not sophisticated: unlike other dependent type systems, it does not support unification, for example. Nevertheless, it is capable of checking all the case studies and benchmarks in our experiments (Section 5.5) without any type or label annotations, except for top-level definitions.

At a high level, like standard bidirectional type checkers, our type checker operates in an *inference mode* and a *checking mode*. In inference mode, the algorithm infers the type of an expression (bottom-up), while in checking mode, the algorithm checks the expression against an expected type by propagating information to subexpressions as deeply as possible (top-down). Our type checker always starts with checking mode, as all top-level definitions are annotated with their type. Figure 5.9 shows a representative selection of our bidirectional type checking rules, using the inference judgment $\Gamma \vdash \mathsf{e} \Longrightarrow_l \tau \rhd \dot{\mathsf{e}}$ and the checking judgment $\Gamma \vdash \mathsf{e} \Longleftarrow_l \tau \rhd \dot{\mathsf{e}}$, both of which output a fully-elaborated expression $\dot{\mathsf{e}}$ in core Taype and in ANF. As explained previously, the inference judgment generates the type τ as an output, while

$\boxed{\Gamma \vdash \mathsf{e} \Longrightarrow_l \tau \rhd \dot{\mathsf{e}}}$ $\boxed{\Gamma \vdash \mathsf{e} \Longleftarrow_l \tau \rhd \dot{\mathsf{e}}}$

TI-Unit
$$\frac{}{\Gamma \vdash () \Longrightarrow_\bot \mathbb{1} \rhd ()}$$

TI-Asc
$$\frac{\Gamma \vdash \mathsf{e} \Longleftarrow_l \tau \rhd \dot{\mathsf{e}} \qquad \Gamma \vdash \tau :: *}{\Gamma \vdash (\mathsf{e} : \tau) \Longrightarrow_l \tau \rhd \dot{\mathsf{e}}}$$

TC-Infer
$$\frac{\Gamma \vdash \mathsf{e} \Longrightarrow_l \tau' \rhd \dot{\mathsf{e}} \qquad \tau \equiv \tau'}{\Gamma \vdash \mathsf{e} \Longleftarrow_l \tau \rhd \dot{\mathsf{e}}}$$

TC-Abs
$$\frac{\mathsf{x} :_{l_1} \tau_1, \Gamma \vdash \mathsf{e} \Longleftarrow_l \tau_2 \rhd \dot{\mathsf{e}} \qquad \Gamma \vdash \tau_1 :: *}{\Gamma \vdash \lambda \mathsf{x} \Rightarrow \mathsf{e} \Longleftarrow_l \Pi \mathsf{x} :_{l_1} \tau_1 , \tau_2 \rhd \lambda \mathsf{x} :_{l_1} \tau_1 \Rightarrow \dot{\mathsf{e}}}$$

TI-App
$$\frac{\Gamma \vdash \mathsf{e}_2 \Longrightarrow_l \Pi \mathsf{x} :_{l_1} \tau_1 , \tau_2 \rhd \dot{\mathsf{e}}_2 \qquad \Gamma \vdash \mathsf{e}_1 \Longleftarrow_{l_1'} \tau_1 \rhd \dot{\mathsf{e}}_1 \qquad \dot{\mathsf{e}}_1 \; {}_{l_1'}\!\rhd_{l_1} \dot{\mathsf{e}}_1'}{\Gamma \vdash \mathsf{e}_2 \;\; \mathsf{e}_1 \Longrightarrow_l [\mathsf{e}_1/\mathsf{x}]\tau_2 \rhd \begin{array}{l} \mathtt{let}\ \mathsf{x}_2 :_l \Pi \mathsf{x} :_{l_1} \tau_1 , \tau_2 \ = \ \dot{\mathsf{e}}_2 \ \mathtt{in} \\ \mathtt{let}\ \mathsf{x}_1 :_{l_1} \tau_1 \ = \ \dot{\mathsf{e}}_1' \ \mathtt{in} \\ \mathsf{x}_2 \;\; \mathsf{x}_1 \end{array}}$$

TI-Pair
$$\frac{\Gamma \vdash \mathsf{e}_1 \Longrightarrow_{l_1} \tau_1 \rhd \dot{\mathsf{e}}_1 \qquad \Gamma \vdash \mathsf{e}_2 \Longrightarrow_{l_2} \tau_2 \rhd \dot{\mathsf{e}}_2 \qquad l = l_1 \sqcup l_2 \qquad \dot{\mathsf{e}}_1 \; {}_{l_1}\!\rhd_{l} \dot{\mathsf{e}}_1' \qquad \dot{\mathsf{e}}_2 \; {}_{l_2}\!\rhd_{l} \dot{\mathsf{e}}_2'}{\Gamma \vdash (\mathsf{e}_1 , \mathsf{e}_2) \Longrightarrow_l \tau_1 \times \tau_2 \rhd \begin{array}{l} \mathtt{let}\ \mathsf{x}_1 :_l \tau_1 \ = \ \dot{\mathsf{e}}_1' \ \mathtt{in} \\ \mathtt{let}\ \mathsf{x}_2 :_l \tau_2 \ = \ \dot{\mathsf{e}}_2' \ \mathtt{in} \\ (\mathsf{x}_1 , \mathsf{x}_2) \end{array}}$$

TI-If
$$\frac{\Gamma \vdash \mathsf{e}_0 \Longleftarrow_\bot \mathbb{B} \rhd \dot{\mathsf{e}}_0 \qquad \Gamma \vdash \mathsf{e}_1 \Longrightarrow_{l_1} \tau_1 \rhd \dot{\mathsf{e}}_1 \qquad \Gamma \vdash \mathsf{e}_2 \Longrightarrow_{l_2} \tau_2 \rhd \dot{\mathsf{e}}_2 \qquad l = l_1 \sqcup l_2 \qquad \Gamma \vdash \mathtt{if}\ \ \mathsf{e}_0\ \ \mathtt{then}\ \ [\cdot ; \cdot \vdash \tau_1]\ \ \mathtt{else}\ \ [\cdot ; \cdot \vdash \tau_2] \rhd \dot{\tau}}{\Gamma \vdash \mathtt{if}\ \ \mathsf{e}_0\ \ \mathtt{then}\ \ \mathsf{e}_1\ \ \mathtt{else}\ \ \mathsf{e}_2 \Longrightarrow_l \dot{\tau} \rhd \ldots}$$

TI-Match
$$\frac{\mathtt{data}\ \ \mathsf{T} \ = \ \overline{\mathsf{C}\ \ \tau} \in \Sigma \qquad \Gamma \vdash \mathsf{e}_0 \Longrightarrow_\bot \mathsf{T} \rhd \dot{\mathsf{e}}_0 \qquad \forall i.\ \mathsf{x} :_\bot \tau_i, \Gamma \vdash \mathsf{e}_i \Longrightarrow_{l_i} \tau_i' \rhd \dot{\mathsf{e}}_i \qquad l = \sqcup l_i \qquad \Gamma \vdash \mathtt{match}\ \ \mathsf{e}_0\ \ \mathtt{with}\ \ \mathsf{C}\ \ \mathsf{x} \Rightarrow [\cdot ; \mathsf{x} :_\bot \tau \vdash \tau']}{\Gamma \vdash \mathtt{match}\ \ \mathsf{e}_0\ \ \mathtt{with}\ \ \overline{\mathsf{C}\ \ \mathsf{x} \Rightarrow \mathsf{e}} \Longrightarrow_l \dot{\tau} \rhd \ldots}$$

Figure 5.9. Selected surface Taype bidirectional typing rules

$$\boxed{\Gamma \vdash \mathcal{E}[\overline{\Delta; \mathcal{P} \vdash \tau}] \rhd \dot{\tau}}$$

$$\textsc{DI-NoDep}\ \frac{\forall i.\ \tau_i \equiv \dot{\tau} \qquad \Delta, \Gamma \vdash \dot{\tau} :: *}{\Gamma \vdash \mathcal{E}[\overline{\Delta; \mathcal{P} \vdash \tau}] \rhd \dot{\tau}} \qquad \textsc{DI-Dep}\ \frac{\forall i.\ \Delta, \mathcal{P}, \Gamma \vdash \tau_i :: *^{\mathsf{O}}}{\Gamma \vdash \mathcal{E}[\overline{\Delta; \mathcal{P} \vdash \tau}] \rhd \mathcal{E}[\overline{\tau}]}$$

$$\textsc{DI-Prod}\ \frac{\Gamma \vdash \mathcal{E}[\overline{\Delta; \mathcal{P} \vdash \tau_1}] \rhd \dot{\tau}_1 \qquad \Gamma \vdash \mathcal{E}[\overline{\Delta; \mathcal{P} \vdash \tau_2}] \rhd \dot{\tau}_2}{\Gamma \vdash \mathcal{E}[\overline{\Delta; \mathcal{P} \vdash \tau_1 \times \tau_2}] \rhd \dot{\tau}_1 \times \dot{\tau}_2}$$

$$\textsc{DI-Pi}\ \frac{\Gamma \vdash \mathcal{E}[\overline{\Delta; \mathcal{P} \vdash \tau_1}] \rhd \dot{\tau}_1 \qquad \Gamma \vdash \mathcal{E}[\overline{\mathtt{x} :_l \tau_1, \Delta; \mathcal{P} \vdash \tau_2}] \rhd \dot{\tau}_2}{\Gamma \vdash \mathcal{E}[\overline{\Delta; \mathcal{P} \vdash \Pi \mathtt{x} :_l \tau_1 , \tau_2}] \rhd \Pi \mathtt{x} :_l \dot{\tau}_1 , \dot{\tau}_2}$$

Figure 5.10. Selected inference rules for dependent contexts

the checking judgment takes τ as an input. For simplicity, the rules presented in Figure 5.9 always infer the labels, but our implementation also checks labels bidirectionally; we will discuss how labels are handled shortly. Our implementation also includes a bidirectional kind checker that generates fully-elaborated types in ANF.

As it is standard, we switch between inference and checking modes using TI-Asc and TC-Infer. To automatically promote expressions, several rules (e.g., TI-Pair and TI-App) use an auxiliary relation $\mathtt{e}\ {}_{l_1}\!\rhd_{l_2}\ \dot{\mathtt{e}}$, which potentially inserts a promotion operation to $\mathtt{e}$ according to its label l_1 and the target label l_2. The definition of this relation is straightforward: $\mathtt{e}\ {}_{l_1}\!\rhd_{l_2}$ `let x = e in` $\uparrow\mathtt{x}$ if $l_1 \sqsubset l_2$, $\mathtt{e}\ {}_{l_1}\!\rhd_{l_2}\ \mathtt{e}$ if $l_1 = l_2$, and fail otherwise.

The main challenge to algorithmic type checking are dependent conditionals and ADT case analysis, specifically inferring their *implicit motives*. Since dependent types in Taype are oblivious types, they are more restricted than the ones in most dependent type systems. To see how, consider the expression: `if x then (`λ`b:`$\widehat{\mathbb{B}}$`⇒`$\widehat{\texttt{if}}$ `b then 1 else 0) else (`λ`n:`$\widehat{\mathbb{Z}}$`⇒`$\widehat{\mathbb{Z}}$`#r n)`. Ignoring labels, the left branch of this conditional has type $\widehat{\mathbb{B}} \to \mathbb{Z}$, while the right one has type $\widehat{\mathbb{Z}} \to \mathbb{Z}$. In many dependent type systems, this expression can simply be typed with `if x then` $\widehat{\mathbb{B}} \to \mathbb{Z}$ `else` $\widehat{\mathbb{Z}} \to \mathbb{Z}$. However, this type is not well-kinded in Taype! The type-level computation, `if` in this case, is only defined over oblivious types, which the types in the branches are clearly not. The correct type of this expression has to be

(`if x then` $\widehat{\mathbb{B}}$ `else` $\widehat{\mathbb{Z}}$) $\to \mathbb{Z}$. The same problem also occurs in dependent ADT pattern matching, and when the branches have product types. Our type checker is equipped with special inference and checking rules for handling these cases. Figure 5.10 shows the auxiliary relation for inferring the well-kinded types that may depend on discriminees, which is invoked in the bidirectional typing rules for dependent conditionals and pattern matching, e.g., TI-If and TI-Match. The rules presented in Figure 5.10 are used in the inference mode, but our implementation also has similar rules for the checking mode. The judgment $\Gamma \vdash \mathcal{E}[\overline{\Delta; \mathcal{P} \vdash \tau}] \rhd \dot{\tau}$ infers a type $\dot{\tau}$ for the dependent context $\mathcal{E}$, under the typing context Γ. The context $\mathcal{E}$ has multiple holes corresponding to the branches of this dependent pattern matching. Each of these holes also has a description consisting of its branch's type τ, the pattern variables with their types $\mathcal{P}$, and a local typing context Δ for collecting arguments in a Π-type. For example, TI-If calls this judgment using the context `if` e_0 `then` $\square$ `else` $\square$ with two branches of types τ_1 and τ_2, and empty pattern variable contexts. In the previous example, these two branch types are $\widehat{\mathbb{B}} \to \mathbb{Z}$ and $\widehat{\mathbb{Z}} \to \mathbb{Z}$ respectively. Starting with an empty local context Δ, DI-Pi decomposes these branch types and infers each component recursively. The argument types $\widehat{\mathbb{B}}$ and $\widehat{\mathbb{Z}}$ are obliviously kinded, so DI-Dep generates a dependent type that combines them using the same dependent context: `if` e_0 `then` $\widehat{\mathbb{B}}$ `else` $\widehat{\mathbb{Z}}$. On the other hand, the return type is simply $\mathbb{Z}$ by DI-NoDep, as both branches have the same return type (according to type equivalence $\equiv$). The side condition of DI-NoDep also ensures the inferred type does not refer to pattern variables, which are not in the scope of this type. The return type may refer to the arguments of a Π-type though, which are collected in the context Δ in the rule DI-Pi. DI-Prod is similar to Di-Pi, although it does not need to modify any contexts.

Our bidirectional type checker also infers leakage labels. In contrast to type annotations, label annotations are not required even for top-level function signatures:[5] they are instead derived from a function *attribute*, which indicates the purpose of a function. A function can be marked as either *section*, *retraction*, or *safe* using the `#[attribute]` syntax, as shown in Figure 5.3. A function without an attribute, such as `elem`, implements the program logic in the conventional fragment of Taype. Such functions and their arguments are always labelled

[5]↑In fact, our surface syntax does not allow users to provide label annotations.

as leaky, since they have to accept retracted values to work with the recipe from Figure 4.1. Any intermediate labels in the bodies of such functions can also be reliably inferred to be leaky, as these functions do not mention oblivious types or public views directly. As a result, a programmer can write the functionality as in a conventional functional language. On the other hand, a function annotated with `#[section]`, e.g., $\widehat{\texttt{list\#s}}$, defines a section function. Its public view argument obviously has a safe label, while its data argument (e.g., `list`) has a leaky label. A section function itself is safe, signaling that all potential leaks have been patched. Conversely, a function, annotated with `#[retraction]`, labels its arguments as safe, but itself has leaky label. Lastly, functions annotated with `#[safe]`, e.g., $\widehat{\texttt{elem}}$, are secure functions. These constitute the API of a secure library, so their arguments and the functions themselves are assigned safe labels. Note that while the labels in a function's signature are determined by its attribute, any intermediate labels in its body need to be inferred. Similar to types, labels are inferred bidirectionally. When an inferred label is checked against a label, the checker will insert a promotion if the expected label is more restrictive than the inferred label, and reject the program when the expected label is less restrictive than the inferred label, using the auxiliary relation $\mathtt{e}\,{}_{l_1}\!\rhd_{l_2}\,\dot{\mathtt{e}}$.

5.3 Oil and Translation

This section describes the OADT intermediate language, Oil, and its translation from Taype. The main challenge is how to encode the features of Taype that Oil lacks, including dependent types, leaky operations ($\widehat{\texttt{if}}$ and `tape`), and, most importantly, its tape semantics.

5.3.1 Syntax, Semantics and Type System

Figure 5.11 shows the syntax of Oil. It is mostly a standard ML-style language with rank-1 polymorphism, extended with an *oblivious array* type and its operations. An oblivious array $\mathcal{A}$ is essentially a "buffer" holding all the private data in a joint computation. The elements of this array are the oblivious representation (usually encrypted values) of members of some fixed finite field. To remain agnostic to the underlying cryptographic protocol, Oil does not place any restrictions on the oblivious representation or the finite field, so the array

e	::=		Expressions:
		`() | b | n | x`	literals and variables
		`e⊕e | e⊞e`	(oblivious) integer operations
		`λx⇒e`	function abstraction
		`let x = e in e`	let binding
		`e e | C e`	function and constructor applications
		`if e then e else e`	conditional
		`mux e e e`	atomic conditional
		`(e,e)`	pair
		`match e with (x,x)⇒e`	product elimination
		`match e with C x⇒e`	ADT elimination
		`𝔹#s e | ℤ#s e`	primitive sections
		`𝒜(e) | e⧺e | e(e,e)`	oblivious array operations
		...	size (ℕ) operations omitted
τ	::=		Types
		`𝟙 | 𝔹 | ℤ`	base types
		`𝒜`	oblivious array
		`ℕ`	size type
		`α`	type variable
		`T`	ADT variable
		`τ×τ`	product type
		`τ→τ`	function type
D	::=		Global Definitions:
		`data T[ᾱ] = C τ`	algebraic data type definition
		`fn x[ᾱ]:τ = e`	(recursive) function definition

Figure 5.11. Oil source syntax

can hold the encryption of bits, or shared secrets of 64-bit integers, for example. Conceptually, each array element is simply an oblivious integer that encodes a piece of the private data, such as an oblivious integer, the tag of an oblivious injection, or an oblivious boolean. Programs create an array of size `n` using `𝒜(n)`, concatenate two arrays using ⧺, and take a slice of `n` elements starting at offset `m` in array `a` via `a(m,n)`.

Like Taype, Oil includes oblivious operations, but these operations are restricted to take and produce oblivious arrays, as this is the *only* oblivious type in Oil. The section operations for base types, `𝔹#s` and `ℤ#s`, for example, return a singleton array containing the "encrypted" result.

Types and global definitions are also standard, but Oil also includes the *size type*, $\mathbb{N}$, for array offsets and lengths. Oil has a standard CBV semantics and type system. The semantics of array operations that use out-of-bound indices (e.g., slicing) is undefined: this should never happen if translated from a well-typed Taype program.

5.3.2 Translating from Taype to Oil

Our translation from Taype to Oil is syntax- and type-directed, and uses the leakage label to identify and repair potential leaks in the program. The translation assumes the source program is in *administrative normal form* (ANF), restricting the brown-colored expressions `e` in Figure 5.6 to be variables. The algorithm roughly consists of three components: translating Taype types to Oil types (Figure 5.12), translating Taype expressions to Oil expressions (Figure 5.16 and Figure 5.17), and translating Taype oblivious types to Oil expressions of the size type (Figure 5.18). The full set of these rules are included in Appendix A.3.

Translating Types

Figure 5.12 shows the translation of a Taype type τ to an Oil type, guided by a leakage label l. With the $\bot$ label, public types are translated as they are or congruently, as expected. Oblivious types, in contrast, are always converted to an oblivious array in Oil. The rich typing information of an oblivious type is not thrown away however: as we shall see, this information is used to implement oblivious array operations. Dependent function types are translated to their nondependent counterpart, with the label on the parameter type dictating its translation.

The translation of types under the $\top$ label is more involved. To understand why, recall that an expression with this label may contain a potentially leaky subexpression which should be repaired via `tape`. Thus, its Oil counterpart must be equipped with a similar mechanism capable of patching leaks. Our solution is to explicitly capture the insecure operations associated with a particular leaky type in its Oil representation, and to insert repairs for each kind of leak when translating a leaky expression. We call this first component a *leaky representation*. As an example, an integer expression can have three kinds of leaks:

$\boxed{[\![\tau]\!]_\bot}$

$$[\![\mathbb{B}]\!]_\bot = \mathbb{B} \qquad [\![\mathbb{Z}]\!]_\bot = \mathbb{Z} \qquad [\![\mathtt{T}]\!]_\bot = \mathtt{T} \qquad [\![\tau_1 \times \tau_2]\!]_\bot = [\![\tau_1]\!]_\bot \times [\![\tau_2]\!]_\bot$$

$$[\![\Pi \mathtt{x}:_\iota \tau_1, \tau_2]\!]_\bot = [\![\tau_1]\!]_\iota \to [\![\tau_2]\!]_\bot \qquad [\![\mathbb{1}]\!]_\bot = [\![\widehat{\mathbb{B}}]\!]_\bot = [\![\widehat{\mathbb{Z}}]\!]_\bot = [\![\tau_1 \mathbin{\widehat{\times}} \tau_2]\!]_\bot = [\![\tau_1 \mathbin{\widehat{+}} \tau_2]\!]_\bot = \mathcal{A}$$

$$[\![\widehat{\mathtt{T}}\ \mathtt{e}]\!]_\bot = [\![\mathtt{if}\ \ldots]\!]_\bot = [\![\mathtt{let}\ \ldots]\!]_\bot = [\![\mathtt{match}\ \ldots]\!]_\bot = \mathcal{A}$$

$\boxed{[\![\tau]\!]_\top}$

$$[\![\mathbb{B}]\!]_\top = \widetilde{\mathbb{B}} \qquad [\![\mathbb{Z}]\!]_\top = \widetilde{\mathbb{Z}} \qquad [\![\mathtt{T}]\!]_\top = \widetilde{\mathtt{T}} \qquad [\![\tau_1 \times \tau_2]\!]_\top = [\![\tau_1]\!]_\top \mathbin{\widetilde{\times}} [\![\tau_2]\!]_\top$$

$$[\![\Pi \mathtt{x}:_\iota \tau_1, \tau_2]\!]_\top = [\![\tau_1]\!]_\iota \to [\![\tau_2]\!]_\top \qquad [\![\mathbb{1}]\!]_\top = [\![\widehat{\mathbb{B}}]\!]_\top = [\![\widehat{\mathbb{Z}}]\!]_\top = [\![\tau_1 \mathbin{\widehat{\times}} \tau_2]\!]_\top = [\![\tau_1 \mathbin{\widehat{+}} \tau_2]\!]_\top = \widetilde{\mathcal{A}}$$

$$[\![\widehat{\mathtt{T}}\ \mathtt{e}]\!]_\top = [\![\mathtt{if}\ \ldots]\!]_\top = [\![\mathtt{let}\ \ldots]\!]_\top = [\![\mathtt{match}\ \ldots]\!]_\top = \widetilde{\mathcal{A}}$$

Figure 5.12. Rules for translating core TAYPE types to OIL types

it could be a retraction of a secure integer $\widehat{\mathbb{Z}}\mathtt{\#r}$, it could be a leaky conditional $\widehat{\mathtt{if}}$, or it could be the promotion of a plaintext integer $\uparrow$. The corresponding leaky representation, $\widetilde{\mathbb{Z}}$, is shown in Figure 5.5, and contains a constructor for each of these cases. As every leaky type can leak information via $\uparrow$ and $\widehat{\mathtt{if}}$, all leaky representations should be equipped with a reified form of these leaky expressions. Thus, every leaky representation (with its safe counterpart) forms a *leaky structure*, with operations `prom` and $\widehat{\mathtt{if}}$ for $\uparrow$ and $\widehat{\mathtt{if}}$ respectively. From an implementation perspective, the leaky structure operations define a *typeclass*, so we call a particular `prom` and $\widehat{\mathtt{if}}$ *instances* of this typeclass. As one example, the constructors of $\widetilde{\mathbb{Z}}$ trivially provide the necessary instances. As another example, Figure 5.5 also shows $\widetilde{\mathcal{A}}$, the leaky representation of an oblivious array; its leaky instance is similarly defined by the two constructors of this type. The leaky representation of function types, presented in Figure 5.13, is slightly more complicated. Its two operations are essentially outsourced to the leaky instances of the codomain type β ($\mathtt{prom}_\beta$ and $\widehat{\mathtt{if}}_\beta$), taken as an extra argument that will be resolved at callsites.

In general, the $\widehat{\mathtt{if}}$ instances are usually constructors, as a leaky conditional needs to be irreducible to avoid leaking its private condition. The promotion instances are also constructors

```
fn prom→ [α β β̃] : (β → β̃) → (α → β) → (α → β̃) =
  λprom_β f x ⇒ prom_β (f x)

fn if̂→ [α β̃] : (𝒜 → β̃ → β̃ → β̃) →
                 𝒜 → (α → β̃) → (α → β̃) → (α → β̃) =
  λif̂_β b̂ f̃_1 f̃_2 x ⇒ if̂_β b̂ (f̃_1 x) (f̃_2 x)
```

Figure 5.13. Leaky structures for function types

```
data list = Nil | Cons ℤ list
data ~list = ~Nil | ~Cons ℤ̃ ~list | prom_list list | if̂_list 𝒜 ~list ~list

fn ~match_list [γ̃] : (𝒜 → γ̃ → γ̃ → γ̃) → ~list →
                      γ̃ → (ℤ̃ → ~list → γ̃) → γ̃ =
  λif̂_γ x̃s f_1 f_2 ⇒
    match x̃s with
    | ~Nil ⇒ f_1
    | ~Cons x̃ x̃s′ ⇒ f_2 x̃ x̃s′
    | prom_list xs ⇒
      match xs with
      | Nil ⇒ f_1
      | Cons x xs′ ⇒ f_2 (prom_ℤ x) (prom_list xs′)
    | if̂_list b̂ x̃s_1 x̃s_2 ⇒
      if̂_γ b̂ (~match_list if̂_γ x̃s_1 f_1 f_2) (~match_list if̂_γ x̃s_2 f_1 f_2)
```

Figure 5.14. Leaky structures for `list`s

in our translation, although in general they need not be.[6] Of course, our translation must also explain how to use leaky values, i.e., how to interpret the corresponding *elimination forms* of τ. To illustrate this, consider the leaky structure for `list` shown in Figure 5.14. The leaky representation of lists includes constructors for `Cons` and `Nil`, i.e., the introduction forms of `list`. Its leaky elimination form, $\widetilde{\texttt{match}}_{\texttt{list}}$, is straightforward: the $\texttt{prom}_{\texttt{list}}$ branch promotes the arguments of each constructor before applying the "alternative functions", and

[6]↑Intuitively, these two instances are generated "for free", though not in the algebraic sense. The only free leaky structure is the one for oblivious arrays $\widetilde{\mathcal{A}}$.

```
data T = C ⟦τ⟧⊥
data T̃ = C̃ ⟦τ⟧⊤ | promT T | ifT 𝒜 T̃ T̃

fn matchT [γ̃] : (𝒜 → γ̃ → γ̃ → γ̃) → T̃ → (⟦τ⟧⊤ → γ̃) → γ̃ =
  λifγ x̃ f̄ ⇒
    match x̃ with
    | C̃ x ⇒ f x
    | promT x ⇒ match x with C x ⇒ f (prom(τ) x)
    | ifT b̂ x̃1 x̃2 ⇒ ifγ b̂ (matchT ifγ x̃1 f̄) (matchT ifγ x̃2 f̄)
```

Figure 5.15. Generating leaky ADT definitions

the $\widehat{\texttt{if}}_{\texttt{list}}$ branch essentially encodes the tape semantics rule S-OIF, specialized to the leaky context of `match` expressions, `match` $\square$ `with` $\overline{\texttt{C x⇒e}}$.

A similar recipe is used to derive the leaky representation and its associated functions for other types: the introduction forms are encoded as constructors with the $\widehat{\texttt{if}}$ and `prom` instances, and the elimination forms capture the idea of distributing the corresponding leaky context into the $\widehat{\texttt{if}}$ branches and how $\uparrow$ interacts with this context. While the leaky structures of builtin and arrow types are defined in the OIL prelude, the ones for user-defined ADTs are generated using the algorithm in Figure 5.15. This is how the leaky definition of `list` in Figure 5.14 was generated, for example. An ADT's introduction forms are its constructors, so the leaky representation just renames them, with the constructor argument types translated with label $\top$. The $\widetilde{\texttt{match}}_{\texttt{T}}$ function encodes the elimination form of ADTs, using a list of functions corresponding to branches of a case expression. The $\texttt{prom}_{\texttt{T}}$ branch relies on the instance resolution procedure `prom`$(\cdot)$ to promote constructor arguments.

Translating Expressions

We now present our translation from TAYPE to OIL expressions. As with our translation of types, the translation of expressions is given as a judgment $\Gamma \vdash \texttt{e} \rightsquigarrow_l \dot{\texttt{e}}$, that is indexed by a leakage label l which guides the translation. Figure 5.16 illustrates how l drives the translation of standard constructs: if l identifies an expression as safe, it is simply translated congruently. On the other hand, if an expression is marked as leaky, the translation relies on the leaky

$\boxed{\Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}}$

TR-SecInt

$$\Gamma \vdash \widehat{\mathbb{Z}}\mathtt{\#s\ x} \rightsquigarrow_l \begin{cases} \widehat{\mathbb{Z}}\mathtt{\#s\ x} & \text{if } l = \bot \\ \widetilde{\mathbb{Z}\mathtt{\#s}}\ \mathtt{x} & \text{if } l = \top \end{cases}$$

TR-Pair

$$\Gamma \vdash (\mathtt{x}_1, \mathtt{x}_2) \rightsquigarrow_l \begin{cases} (\mathtt{x}_1, \mathtt{x}_2) & \text{if } l = \bot \\ \widetilde{\mathtt{pair}}\ \mathtt{x}_1\ \mathtt{x}_2 & \text{if } l = \top \end{cases}$$

TR-App

$$\frac{}{\Gamma \vdash \mathtt{x}_2\ \mathtt{x}_1 \rightsquigarrow_l \mathtt{x}_2\ \mathtt{x}_1}$$

TR-Abs

$$\frac{\mathtt{x} :_{l_1} \tau_1, \Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}}{\Gamma \vdash \lambda \mathtt{x} :_{l_1} \tau_1 \Rightarrow \mathtt{e} \rightsquigarrow_l \lambda \mathtt{x} \Rightarrow \dot{\mathtt{e}}}$$

TR-If

$$\frac{\mathtt{x}_0 :_{l_0} \mathbb{B} \in \Gamma \qquad \Gamma \vdash \mathtt{e}_1 \rightsquigarrow_l \dot{\mathtt{e}}_1 \qquad \Gamma \vdash \mathtt{e}_2 \rightsquigarrow_l \dot{\mathtt{e}}_2}{\Gamma \vdash \mathtt{if}_\tau\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \rightsquigarrow_l \begin{cases} \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \dot{\mathtt{e}}_1\ \mathtt{else}\ \dot{\mathtt{e}}_2 & \text{if } l_0 = \bot \\ \widetilde{\mathtt{if}}\ \widehat{\mathtt{if}}(\tau)\ \mathtt{x}_0\ \dot{\mathtt{e}}_1\ \dot{\mathtt{e}}_2 & \text{if } l_0 = \top \end{cases}}$$

Figure 5.16. Selected rules for translating core TAYPE standard expressions to OIL expressions

context of the expression to patch any leaks. This strategy can be seen in the TR-SecInt rule: using this rule to translate a leaky $\widehat{\mathbb{Z}}\mathtt{\#s\ e}$ expression delegates any repairs to $\widetilde{\mathbb{Z}\mathtt{\#s}}$. Translating lambda abstractions (TR-Abs) and applications (TR-App) is straightforward. TR-Pair shows why we require uniform labels in subexpressions: the components of a pair marked as leaky must also be leaky, as $\widetilde{\mathtt{pair}}$ takes the leaky representations as arguments, similar to $\widetilde{\mathtt{Cons}}$ from Figure 5.14. The translation of `if` (TR-If) differs from the other rules in that the label of its discriminee dictates when its leaky counterpart is used, rather than the label of the whole expression. To see why, recall the typing rule T-IfNoDep from Figure 5.8: if the label of the discriminee is $\top$, the label of the whole expression must also be $\top$. On the other hand, we do allow the discriminee to be non-leaky, even if the whole expression is leaky. In this case, we simply use the standard `if` statement, as leaks can only occur in a subexpression. A similar strategy applies when translating `match`. The TR-If rule illustrates why we annotate conditionals and case statements with their result type τ: this type is used to resolve the leaky if instances associated with τ via a call to the metafunction $\widehat{\mathtt{if}}$.

Figure 5.17 presents some of the translation rules for leaky and oblivious constructs. This translation is more involved, as it needs to account for the switch to OIL's oblivious

tp

$\boxed{\Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}}$

TR-Unit

$$\frac{}{\Gamma \vdash \mathtt{()} \rightsquigarrow_\bot \mathcal{A}(\mathtt{0})}$$

TR-OPair

$$\frac{}{\Gamma \vdash \mathtt{[x_1,x_2]} \rightsquigarrow_\bot \mathtt{x_1 \# x_2}}$$

TR-OInj

$$\frac{\Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \widehat{\iota_b}\mathtt{<}\tau_1 \widehat{+} \tau_2\mathtt{>\ x} \rightsquigarrow_\bot \mathtt{ite(b,}\widehat{\mathtt{inl}}\mathtt{,}\widehat{\mathtt{inr}}\mathtt{)\ s_1\ s_2\ x}}$$

TR-RetInt

$$\frac{}{\Gamma \vdash \widehat{\mathbb{Z}}\mathtt{\#r\ x} \rightsquigarrow_\top \mathtt{r}_\mathbb{Z}\ \mathtt{x}}$$

TR-Tape

$$\frac{}{\Gamma \vdash \mathtt{tape\ x} \rightsquigarrow_\bot \widetilde{\mathtt{tape}}\ \mathtt{x}}$$

TR-Promote

$$\frac{\mathtt{x} :_\bot \tau \in \Gamma}{\Gamma \vdash \uparrow\mathtt{x} \rightsquigarrow_\top \mathtt{prom}(\tau)\ \mathtt{x}}$$

TR-OIf

$$\frac{\mathtt{x}_1 :_\top \tau \in \Gamma}{\Gamma \vdash \widehat{\mathtt{if}}\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{x}_1\ \mathtt{else}\ \mathtt{x}_2 \rightsquigarrow_\top \widehat{\mathtt{if}}(\tau)\ \mathtt{x}_0\ \mathtt{x}_1\ \mathtt{x}_2}$$

TR-OPMatch

$$\frac{\mathtt{x}_1 :_\bot \tau_1, \mathtt{x}_2 :_\bot \tau_2, \Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}} \qquad \Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \widehat{\mathtt{match}}\ \mathtt{x}_0\mathtt{:}\tau_1 \widehat{\times} \tau_2\ \mathtt{with\ [x_1,x_2]} \Rightarrow \mathtt{e} \rightsquigarrow_l \begin{array}{l}\mathtt{let\ x_1\ =\ x_0(0,s_1)\ in}\\ \mathtt{let\ x_2\ =\ x_0(s_1,s_2)\ in}\\ \dot{\mathtt{e}}\end{array}}$$

TR-OMatch

$$\frac{\mathtt{x} :_\bot \tau_1, \Gamma \vdash \mathtt{e}_1 \rightsquigarrow_\top \dot{\mathtt{e}}_1 \qquad \mathtt{x} :_\bot \tau_2, \Gamma \vdash \mathtt{e}_2 \rightsquigarrow_\top \dot{\mathtt{e}}_2 \qquad \Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \widehat{\mathtt{match}}_\tau\ \mathtt{x}_0\mathtt{:}\tau_1 \widehat{+} \tau_2\ \mathtt{with\ x} \Rightarrow \mathtt{e}_1 \mid \mathtt{x} \Rightarrow \mathtt{e}_2 \rightsquigarrow_\top \begin{array}{l}\mathtt{let\ tag\ =\ x_0(0,1)\ in}\\ \widehat{\mathtt{if}}(\tau)\ \mathtt{tag\ (let\ x\ =\ x_0(1,s_1)\ in\ \dot{e}_1)}\\ \qquad\qquad \mathtt{(let\ x\ =\ x_0(1,s_2)\ in\ \dot{e}_2)}\end{array}}$$

Figure 5.17. Selected rules for translating core Taype leaky and oblivious expressions to Oil expressions

$\boxed{\Gamma \vdash \tau \rightsquigarrow \mathtt{s}}$

$$\frac{}{\Gamma \vdash \mathbb{1} \rightsquigarrow 0}\ \textsc{TR-UnitT} \qquad \frac{}{\Gamma \vdash \widehat{\mathbb{Z}} \rightsquigarrow 1}\ \textsc{TR-OInt} \qquad \frac{\Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \tau_1 \mathbin{\widehat{\times}} \tau_2 \rightsquigarrow \mathtt{s}_1 + \mathtt{s}_2}\ \textsc{TR-OProd}$$

$$\frac{\Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \tau_1 \mathbin{\widehat{+}} \tau_2 \rightsquigarrow 1 + \mathtt{max}\ \mathtt{s}_1\ \mathtt{s}_2}\ \textsc{TR-OSum} \qquad \frac{}{\Gamma \vdash \widehat{\mathtt{T}}\ \mathtt{x} \rightsquigarrow \widehat{\mathtt{T}}\ \mathtt{x}}\ \textsc{TR-TApp}$$

$$\frac{\Gamma \vdash \mathtt{e} \rightsquigarrow_\perp \dot{\mathtt{e}} \qquad \mathtt{x} :_\perp \tau_1, \Gamma \vdash \tau \rightsquigarrow \mathtt{s}}{\Gamma \vdash \mathtt{let}\ \mathtt{x} :_\perp \tau_1 = \mathtt{e}\ \mathtt{in}\ \tau \rightsquigarrow \mathtt{let}\ \mathtt{x} = \dot{\mathtt{e}}\ \mathtt{in}\ \mathtt{s}}\ \textsc{TR-TLet}$$

Figure 5.18. Selected rules for translating core Taype oblivious types to sizes in Oil

arrays. This is straightforward for simple data types: unit values are simply encoded as an empty array (TR-Unit), while the translation of an oblivious pair simply concatenates the arrays produced by the translation of its two components (TR-OPair). Translating the destructor for oblivious pairs is more interesting (TR-OPMatch), as it needs to extract each component from a flat array. To see how this is possible, observe that the "size" of an oblivious value is determined by its type, otherwise we risk leaking private information through this side-channel: thus, we can determine the location of each component of a pair based solely on their types. We do so via an auxiliary relation, $\Gamma \vdash \tau \rightsquigarrow \mathtt{s}$, given in Figure 5.18, which translate Taype types to Oil size expressions.

The translation of oblivious injections provide another example of how this relation is used. The TR-OInj rule relies on the auxiliary function (and a similar right injection function) shown in Figure 5.19. This function takes as input the sizes of the left and right components and the injection payload, and produces an oblivious array containing the tag and payload, padding it out to the size of the larger component to avoid leaking information through its representation.

```
fn inl : ℕ → ℕ → 𝒜 → 𝒜 =
  λm n â ⇒
    let tag = 𝔹#s true in
    let payload =
          if n ≤ m then â
          else â ++ 𝒜(n−m)
    in tag ++ payload
```

Figure 5.19. Oblivious injection

$\boxed{\mathtt{D} \rightsquigarrow \dot{\mathtt{D}}}$

TR-OADT

$$\frac{\mathtt{x} :_{\perp} \tau', \Gamma \vdash \tau \rightsquigarrow \mathtt{s}}{\mathtt{obliv}\ \widehat{\mathtt{T}}\ (\mathtt{x}{:}\tau') = \tau \rightsquigarrow \mathtt{fn}\ \widehat{\mathtt{T}}{:}[\![\tau']\!]_{\perp}{\to}\mathbb{N} = \lambda \mathtt{x} \Rightarrow \mathtt{s}}$$

TR-Fun

$$\frac{\cdot \vdash \mathtt{e} \rightsquigarrow_{l} \dot{\mathtt{e}}}{\mathtt{fn}\ \mathtt{x}{:}_{l}\tau = \mathtt{e} \rightsquigarrow \mathtt{fn}\ \mathtt{x}{:}[\![\tau]\!]_{l} = \dot{\mathtt{e}}}$$

Figure 5.20. Selected rules for translating core Taype definitions to Oil definitions

$\boxed{\widehat{\mathtt{if}}(\tau)}$

$$\widehat{\mathtt{if}}(\mathcal{A}) = \widehat{\mathtt{if}}_{\mathcal{A}} \qquad \widehat{\mathtt{if}}(\mathbb{B}) = \widehat{\mathtt{if}}_{\mathbb{B}} \qquad \widehat{\mathtt{if}}(\mathbb{Z}) = \widehat{\mathtt{if}}_{\mathbb{Z}} \qquad \widehat{\mathtt{if}}(\mathtt{T}) = \widehat{\mathtt{if}}_{\mathtt{T}} \qquad \widehat{\mathtt{if}}(\alpha{\times}\beta) = \widehat{\mathtt{if}}_{\times}$$

$$\widehat{\mathtt{if}}(\alpha{\to}\beta) = \widehat{\mathtt{if}}_{\to}\ \widehat{\mathtt{if}}(\beta)$$

$\boxed{\mathtt{prom}(\tau)}$

$$\mathtt{prom}(\mathcal{A}) = \mathtt{prom}_{\mathcal{A}} \qquad \mathtt{prom}(\mathbb{B}) = \mathtt{prom}_{\mathbb{B}} \qquad \mathtt{prom}(\mathbb{Z}) = \mathtt{prom}_{\mathbb{Z}} \qquad \mathtt{prom}(\mathtt{T}) = \mathtt{prom}_{\mathtt{T}}$$

$$\mathtt{prom}(\alpha{\times}\beta) = \mathtt{prom}_{\times} \qquad \mathtt{prom}(\alpha{\to}\beta) = \mathtt{prom}_{\to}\ \mathtt{prom}(\beta)$$

Figure 5.21. Resolving leaky instances

For example, the translation of $\widehat{\mathtt{inl}}\langle\widehat{\mathbb{Z}}\hat{+}\widehat{\mathbb{Z}}\hat{\times}\widehat{\mathbb{Z}}\rangle$ `[2]` computes to oblivious array `[1,2,0]`, while $\widehat{\mathtt{inr}}\langle\widehat{\mathbb{Z}}\hat{+}\widehat{\mathbb{Z}}\hat{\times}\widehat{\mathbb{Z}}\rangle$ `[[3],[4]]` computes to `[0,3,4]`.

The remaining rules in the figure adopt similar strategies; relying on a combination of leaky structures to patch up leaky constructs and the size relation to bridge the gap between oblivious types in Taype and oblivious arrays in Oil: the rule for `tape` (TR-Tape), for example, simply delegates the repair to $\widetilde{\mathtt{tape}}$, from Figure 5.5, which encodes the tape rules S-TapeOIf and S-TapeProm. Similarly, the TR-OMatch uses the size to extract the tag of a sum type, before processing both branches with a leaky $\widehat{\mathtt{if}}$ expression which eventually discards the unused branch. Note that the payload `x` extracted from the injection in the "wrong" branch always uses the right size for that type: when matching on the previous example, `x` will be `[2,0]` in the second branch.

The translation of top-level definitions is straightforward; Figure 5.20 provides the rules for oblivious ADTs and functions of this translation. The elided translation of ADTs simply relies on the generation algorithm from Figure 5.15. The resolution procedures, $\widehat{\mathtt{if}}$ and `prom`, are

also straightforward, shown in Figure 5.21. Most instances are resolved to the corresponding definitions in the Oil prelude, or to the generated constructors for ADTs. Resolving function type instances also requires resolving the instances of a function's codomain, as suggested by Figure 5.13. Note that `prom` and $\widehat{\texttt{if}}$ in Figure 5.21 take an Oil type, while Taype types are given to the calls to these meta-functions in the translation rules; we implicitly apply $[\![\cdot]\!]_\perp$ (Figure 5.12) to convert Taype types to Oil types for the leaky instance resolution.

Our translation algorithm is guaranteed to terminate, even when the source program does not. The reason can be seen in our translation rules, as every (mutually) recursive call to the translation judgment is applied to a structurally smaller core Taype sub-expression. The algorithm enjoys a stronger totality property: translation of a well-typed core Taype program never fails, i.e., a well-typed program satisfies all the side-conditions of the translation rules:

Theorem 5.3.1 (Totality of Translation)**.** *If* $\Gamma \vdash \mathtt{e} :_l \tau$ *and* $\mathtt{e}$ *is in ANF, then there exists an Oil expression* $\dot{\mathtt{e}}$ *such that* $\Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}$.

The proof (and an analogous theorem for the type-to-size translation) is given in Appendix A.4.

5.3.3 Translation for Conceal and Reveal Phases

Secure multiparty computations typically consist of three phases: a *conceal phase*, an *oblivious computation phase*, and a *reveal phase*. In the conceal phase, private data owners "encrypt" and share their data before the core computation takes place, while the oblivious output is revealed to all (or the privileged) parties in the reveal phase. In order to provide a complete solution, we also produce secure implementations of these two phases. Thankfully, section and retraction functions provide templates for concealing and revealing private data. Our toolchain thus translates section and retraction functions to special versions that implement each phase.

Translating the retraction functions needed for the reveal phase is simple: we simply make all the leaky operations "leak" by renaming all leaky operations in a retraction function, and link them to the revealing versions. For example, $\widehat{\mathbb{Z}}$`#r` is renamed to `Reveal.`$\widehat{\mathbb{Z}}$`#r` (in an

OCaml module named `Reveal`). The retraction functions themselves are also renamed so client programs can use them to reveal the results of the computation.

Translating the section functions needed for the conceal phase is more involved. The main problem is that, unlike the core computation, only the private data owner can run the conceal function, as other parties do not have the data. But many MPC protocols, e.g., ones based on secret-sharing, require all parties to help create the encryption of the private information, so this has to be done synchronously. In our setting, since private data is encoded as oblivious arrays, all parties have to encrypt the elements of the same index at the same time. For example, during the conceal phase, if Alice is encrypting the third element of the array, Bob needs to do the same. However, this is not trivial to enforce: how does Bob know which element Alice is currently dealing with, when he does not have the data? Naturally, Bob may only construct the oblivious array from left to right, which means that Alice needs to do the same. Our implementation conceals private data in two steps. First, the private data owners (e.g., Alice) run the section functions *locally* using a plaintext backend for cryptographic operations (Section 5.4), resulting in an "oblivious" array whose elements are not encrypted, i.e., a plaintext array. Then, all parties (jointly) encrypt the input array from left to right, using the underlying cryptographic protocol, and obtain the actual oblivious array needed for the oblivious computation phase.

5.4 Implementation

We have implemented the above approach as a compiler that takes as input a Taype program containing the functions to be computed (as well as any auxiliary functions), the public views, and section and retraction functions. After type checking these pieces, our toolchain produces OCaml implementations of the conceal and reveal phases, as well as an OCaml implementation of the multiparty computation, all of which are specialized to the desired public view. The output programs are clients of a module that provides an implementation of Oil's oblivious arrays and oblivious operations. Linking the generated programs with an implementation of this interface, or *driver*, produces a library that a programmer can use to build a secure application: they simply gather the private data,

"encrypt" the data using the generated conceal functions, call the multiparty functionality from the library, and finally reveal the result using the generated reveal functions. As the calculator case study in the next section demonstrates, programmers can also implement multi-round computation by chaining together calls to this library.

Our current implementation features two drivers: a plaintext driver and a cryptography-backed driver. The plaintext driver computes its results in the clear, and is intended for testing purposes and for establishing a performance baseline without any cryptographic overhead. This driver is also necessary for generating section functions, as explained in the previous section. The cryptographic driver uses the popular open-source EMP toolkit [52] to implement secure computations. This library is based on Yao's Garbled Circuit [1] for semi-honest 2-party MPC. Integrating a new backend into our framework is conceptually simple: the driver just needs to implement an interface consisting of oblivious integer encryption, decryption and its arithmetic. Our EMP toolkit backend consists of boilerplate code for FFI (foreign function interface), for example. Other aspects of the driver, such as array operations, are independent of the cryptographic backends, and can thus be shared among all drivers.

5.4.1 Optimizations

We have implemented two optimizations to improve the performance of generated code.

First, the unoptimized programs produced naively by the strategy in Figure 5.16 and Figure 5.17 can suffer from exponential blowup when leaks are "taped" too late. To see the issue, consider a recursive function `f` from `list` to $\mathbb{Z}$, whose recursive calls on the tail of the input list appear in both branches of a conditional, either directly or indirectly via variables: e.g., `if x then f xs′ else 1 + f xs′`. If `x` is a leaky boolean of the form $\widehat{\texttt{if}}$ `[b] then true else false`, the conditional will be distributed into both branches, resulting in $\widehat{\texttt{if}}$ `[b] then f xs′ else 1 + f xs′`. But `f xs′` also produces a similar $\widehat{\texttt{if}}$, in both branches! For example, one more unrolling of the recursive calls may result in:

```
if [b]
then (if [b′] then f xs″ else 1 + f xs″)
else (if [b′] then 1 + f xs″ else 2 + f xs″)
```

Consequently, compiling `f` results in a $\widehat{\texttt{if}}$ tree that is exponential in the length of the list, causing exponential blowup in both memory and running time.

To ameliorate this problem, we have implemented an optimization that `tape`s leaky expressions earlier (called "early-tape" optimization), by wrapping the whole body of this recursive function with a composition of section and retraction: $\widehat{\mathbb{Z}}$`#r (tape (`$\widehat{\mathbb{Z}}$`#s (f xs)))`. This transformation does not change the semantics of `f`, and the combination of `tape` and $\widehat{\mathbb{Z}}$`#s` forces the $\widehat{\texttt{if}}$ of each recursive call to be securely reduced to a single oblivious integer to avoid building up the $\widehat{\texttt{if}}$ tree. This simple strategy is quite effective for a class of programs which return combinations of primitive types and tuples.

Another deficiency of a naive implementation is that the sizes of oblivious data may be repeatedly computed. For example, the retraction function of the oblivious list with public view `k` requires computing $\widehat{\texttt{list}}$ `k` as a size, but its recursive calls on `k−1` also requires computing $\widehat{\texttt{list}}$ `(k−1)`, while this computation has already been done in the previous iteration. This redundant computation attaches a quadratic term to the complexity of the retraction function. As a result, linear scan of an oblivious list becomes quadratic.

To remove the redundant size computation, our compiler implements a classic technique called *tupling* [53, 54], which merges some recursive functions into a tuple to avoid multiple traversals over the same data. Consider the previous example: the oblivious list retraction function $\widehat{\texttt{list}}$`#r` has type $\mathbb{Z} \to \mathcal{A} \to \widetilde{\texttt{list}}$ in Oil, while the oblivious list function $\widehat{\texttt{list}}$, i.e., the size function in Oil, has type $\mathbb{Z} \to \mathbb{N}$. We create a new recursive function $\widehat{\texttt{list}}$`#r_tupled` of type $\mathbb{Z} \to \mathbb{N} \times (\mathcal{A} \to \widetilde{\texttt{list}})$, whose definition should be equivalent to the tuple of those two functions:

```
list#r_tupled k = (list k, list#r k)
```

The actual recursive definition of this tupled function is obtained by a transformation based on the unfold-simplify-fold rules [55, 56], which eventually replaces the calls to $\widehat{\texttt{list}}$ and $\widehat{\texttt{list}}$`#r` with recursive calls to this function itself. To see how this avoids repetitive size computation, observe that each recursive call to this tupled function on `k−1` returns not only the result of $\widehat{\texttt{list}}$`#r (k−1)` but also $\widehat{\texttt{list}}$ `(k−1)`, meaning that we can calculate $\widehat{\texttt{list}}$ `k` from that incrementally.

5.5 Experiments

Our experiments consist of a set of case studies that showcase the applicability of our approach, summarized in Figure 5.22, and a set of microbenchmarks that examine the empirical benefits of being able to trade off security for performance.

5.5.1 Case Study: Medical Records

Our first collection of case studies are inspired by problems in the healthcare setting, where legal and privacy concerns keep parties from freely sharing their data. These benchmarks use a variety of data structures: patient data is represented as a record with fields for a patient's ID, age, height and weight, a database is encoded as a list of patient records, and a classifier is implemented as a decision tree over health data. The oblivious types for these data structures admit interesting public views: a particular health record may choose to keep either its ID, or medical data (height and weight) secret, as in Figure 3.5; a database adopts the privacy settings (i.e., revealing either ID or medical data) of its individual records, and a decision tree obscures the threshold that a feature is compared against at a given node, but not the overall structure. Using these representations, we have implemented a number of secure computations: biometric matching (minimum euclidean distance) between a single record and a database, calculating the percentage of healthy members of an age group according to the Body Mass Index (BMI), calculating statistics such as mean and variance over multiple private databases, and classifying a patient record using a private decision tree. Figure 5.22 includes each of these programs. Notably, the computation in each of these benchmarks was written without a privacy policy in mind; instead, our compiler took care of enforcing each policy, as encoded by an oblivious type and section and retraction functions.

5.5.2 Case Study: Dating Application

Consider a functionality of matching potential soulmates. Each party owns their private profile with personal information, such as gender, income and education. They also have a private preference for their partner, encoded as predicates over profiles of *both* parties.

Computation	Alice's input	Bob's input	Description
is-taller	record	record	a variant of the millionaire problem, comparing height
is-obese-by-id	database	ID	whether the record of a given ID is obese (according to BMI)
healthy-rate-by-age	database	a range of ages	the percentage of healthy members of an age range (according to BMI)
min-euclidean-distance	database	record	the minimum euclidean distance between a database and a given record
database-analytics	database	database	calculate the mean and variance of the ages over these databases
mean-squared-error	database	database of BMIs	the mean squared error between the estimated BMIs (from Bob) and the actual BMIs (from Alice); the two databases may not contain the same records, and are matched with IDs (similar to joining tables)
decision-tree	decision tree	record	classify a medical record via a private decision tree
dating	profile and predicate	profile and predicate	match the dating preferences of two parties; each party provides a private profile and a preference encoded as a predicate over profiles
secure-calculator	expression and assignment	expression and assignment	a 2-round arithmetic expression evaluation; each party provides a private arithmetic expression and some private variable assignment depending on the round
voting	tabulated votes	tabulated votes	return the candidate with the most votes
k-means	list of vectors	list of vectors	partition vectors using the k-means clustering algorithm

Figure 5.22. Summary of programs used in our case studies

These predicates are expressions with boolean connectives, integer arithmetic and numeric comparisons. For example, one user may stipulate that the sum of both parties' income exceeds a particular amount. The peer matching function takes these private profiles and predicates, and returns a boolean indicating whether they are a good match, by evaluating the predicate expression on the profiles. The private predicates have many potential policies. As predicates are essentially ASTs, participating parties may agree on disclosing only the depth of the predicates, or revealing the AST nodes but not the operands, or even revealing only the boolean connectives but keep the integer expressions secret. In Taype, the peer matching function, its auxiliary functions, and the data types they depend on can be implemented in the conventional way, without knowing the policies. The private matching function can be obtained by composing with the desired policy. Updating policy or updating the matching algorithm can be done independently.

5.5.3 Case Study: Secure Calculator

To showcase our support for computations involving richly structured data, we have implemented a secure interpreter for a simple arithmetic expression language. In this case study, each user provides a private expression and an assignment to some variables. The result is securely computed by evaluating the first party's expression using the second party's assignment; the result of this expression is then used to evaluate the second party's expression, along with the first party's private value, as shown in Figure 5.23. Not only does this case study use a rich data structure for expression, it also shows that we can readily compose the generated library functions to implement a more complex workflow, such as a multi-round computation.

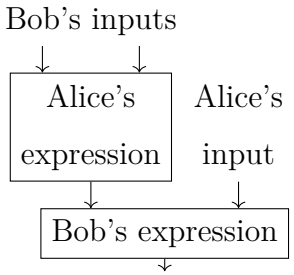


Figure 5.23. Workflow of the secure calculator

5.5.4 Discussion

As mentioned in Chapter 1, in existing frameworks, it is the program's responsibility to enforce the privacy policy. In contrast, our medical records, dating application and secure calculator case studies demonstrate that, in our framework secure functionality can be written in a conventional functional language, agnostic to a particular privacy policy. On the other hand, implementing oblivious types, section and retraction functions is analogous to other common programming tasks: an oblivious type is essentially a different representation of the underlying data type, while section and retraction functions are effectively conversion functions between oblivious and public data types. Importantly, our abstraction allows programmers to write all these "boilerplate" functions once and for all, regardless of a particular target computation.

5.5.5 Microbenchmarks

To evaluate the performance of our compiler,[7] we have built a number of microbenchmarks that showcase the performance tradeoffs between privacy and performance. Our first microbenchmark is a standard classification scenario, where one party wants to classify their private data using a decision tree belonging to another party [57–59]. The data being classified is given as a tuple with eight private integers as features. This experiment considers 4 public views for the decision tree: maximum height, the spine, spine including the feature index of each node, and the whole tree. Note that this last view is not unrealistic: in outsourced secure computation, such as FHE [7], the decision tree owner may perform all computation, independent of the other party. In this scenario, the whole tree can be revealed because the computing party owns it, but the computation should not reveal any information about the other party's data. The definition of the decision tree is shown in Figure 5.24a, together with an "oblivious" version that simply reveals the whole tree. Figure 5.24 also includes views for three other policies. The section and retraction functions for each view are analogous to those in Figure 5.3. For each public view, we test on a small tree of depth 1, and other

[7]↑ All results are averaged across 10 runs, on an M1 MacBook Pro with 16 GB memory. All parties run on the same host with local network communication.

```
// The payload of Leaf is a decision,
// and that of Node is a feature index
// and a threshold.
data tree = Leaf ℤ
          | Node ℤ ℤ tree tree
obliv ⌢tree_all (_ : tree) = 𝟙
```

(a) All information

```
obliv ⌢tree_max (k : ℤ) =
  if k = 0
  then ℤ̂
  else ℤ̂ ∔
       ℤ̂ ×̂ ℤ̂ ×̂ ⌢tree_max (k−1)
           ×̂ ⌢tree_max (k−1)
```

(b) Maximum height

```
data spine = SLeaf
           | SNode spine spine
obliv ⌢tree_spine (s : spine) =
  match s with
  | SLeaf ⇒ ℤ̂
  | SNode l r ⇒
    ℤ̂ ×̂ ℤ̂ ×̂ ⌢tree_spine l ×̂ ⌢tree_spine r
```

(c) Tree spine

```
data spineF = SFLeaf
            | SFNode ℤ spineF spineF
obliv ⌢tree_spineF (s : spineF) =
  match s with
  | SFLeaf ⇒ ℤ̂
  | SFNode _ l r ⇒
    ℤ̂ ×̂ ⌢tree_spineF l ×̂ ⌢tree_spineF r
```

(d) Tree spine with feature indices

Figure 5.24. Definitions of oblivious decision trees with different public views

trees of depth 16. A *full tree* has exponentially many nodes, while an *eighth sparse tree* has roughly 1/8 of the nodes in a full tree, and a *very sparse tree* has only 16 nodes.

Figure 5.25 reports the performance impact of each view on the total run time. The results are as expected: revealing the whole tree results in the best performance, while sharing only the maximum height is quite slow. In the case of maximum height, the number of nodes in the actual decision tree does not affect the performance, as the structure of the tree is kept secret. Knowing both the spine and the feature index of each node improves performance, compared to knowing

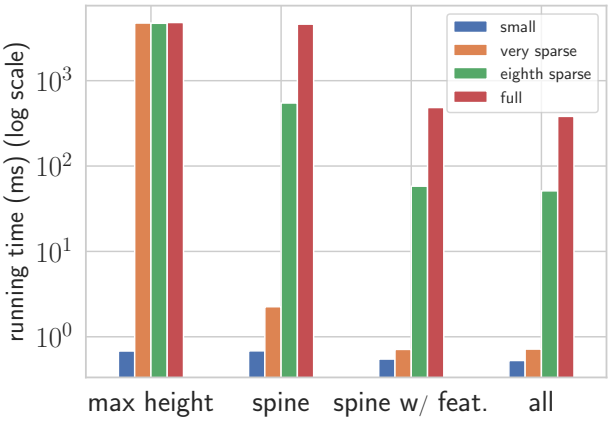
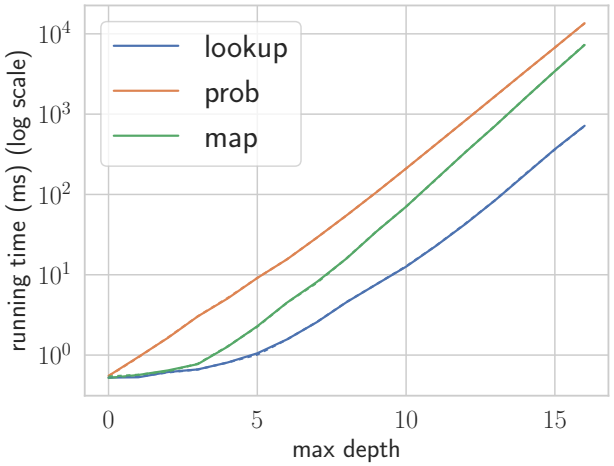
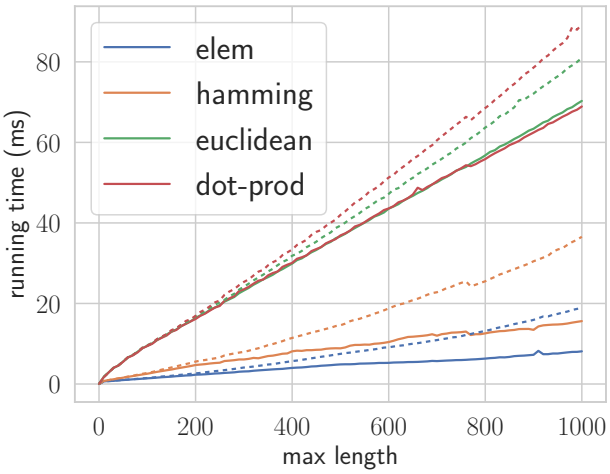


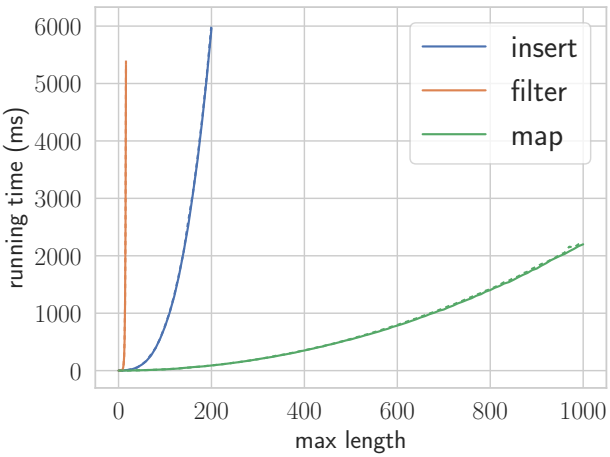
Figure 5.25. Decision tree



(a) Tree example



(b) List examples (primitive)



(c) List examples (complex)

Figure 5.26. Microbenchmarks

only the spine, as the computation does not need to obscure which feature is used at each decision point. When the underlying decision tree is relatively full, leaking more information about its structure does not improve performance, as the program does not need to perform wasted computation to ensure a constant time algorithm. Indeed, the fuller the private tree is, the less is gained by a more permissive public view. Of course, the owner of the tree must ultimately decide if they are willing to reveal how the tree is close to this worst case scenario. Again, the decision algorithm is agnostic of the actual public views, allowing for swapping privacy policies without any changes to the program logic.

We also evaluate the performance of a set of standard operations on trees, using its maximum height as the public view. These benchmarks consist of a membership test, computing the probability of an event given a probability tree diagram, and a map function that adds a private integer to each node in a tree. Figure 5.26a presents the performance results for these benchmarks. Despite the inherently expensive cryptography required by the conservative public view, all the benchmarks finish in under 15 seconds.

Finally, we have implemented a similar set of microbenchmarks for oblivious lists, using the length of the list as the public view. We subdivide these benchmarks into those that return a primitive value (i.e., an integer or boolean), and those that return an oblivious list. Even though the returned lists often reveal too much about the private input, they could be useful as an intermediate result of a bigger computation or as an input to the next round of computation. Figure 5.26b presents the performance for the first category, which includes a membership check, computing of the hamming and euclidean distances between two lists, and taking the dot product of two lists. All of these examples are amenable to the optimization mentioned in the previous section, resulting in reasonable running times. We use a dotted line for results without our tupling optimization, and a solid line for when the optimization is enabled. With tupling is used, their performance is linear in the size of the input list (as it is in the insecure setting). The second category includes insertion into a sorted list, and two higher-order examples: mapping a function that adds a private integer to all the elements of a list, and a filter function that drops all the elements greater than a private integer. Since these examples do not return primitive values, the early-tape optimization does not apply, resulting in slower performance, as Figure 5.26c shows. The tupling optimization does not have much impact, as its gains are overshadowed by the complexity of having to delay repairs to the leaky result values.

5.6 Conclusion

Secure multiparty computation enables different parties to compute functions over private data without leaking extra information, but writing these applications remains challenging. Existing high-level MPC languages require programs to explicitly enforce privacy policies,

making it difficult to update policies and to explore tradeoffs between privacy guarantees and performance. This chapter presented Taype, a language for secure multiparty applications that decouples these concerns. Our experiments feature a diverse set of benchmarks that were written without security policies in mind, and a wide range of security policies that went beyond whether a particular field is "secret or not". Our results demonstrate the performance benefits that can result from being able to easily trade off privacy for performance.

6. TAYPSI: STATIC ENFORCEMENT OF POLICIES

Taype (Chapter 5) decouples privacy policies from programmatic concerns, allowing users to write applications over structured data that are agnostic to any particular privacy policy. To do so, Taype implements a novel form of the *tape semantics* (Chapter 4). This semantics allows insecure operations whose evaluation *could* violate a policy to appear in a program, as long as the results of these operations are *eventually* protected. Under tape semantics, such operations are lazily deferred until it is safe to execute them, effectively *dynamically* "repairing" potential leaks at runtime. Using Taype, programmers can thus build a privacy-preserving version of a standard functional program by composing it with a policy, specified as a *dependent type* equipped with security labels, relying on tape semantics to enforce the policy during execution. Unfortunately, while this enforcement strategy disentangles privacy concerns from program logic, it also introduces considerable overhead for applications that construct or manipulate structured data with complex privacy requirements. Thus, this strategy does not scale to the sorts of complex applications that could greatly benefit from this separation of concerns.

This chapter presents Taypsi, a policy-agnostic language for writing MPC applications that eliminates this overhead by instead transforming a non-secure function into a version that *statically* enforces a user-provided privacy policy. Taypsi extends Taype with a form of dependent sums, which we call Ψ-types, that package together the public and private components of an algebraic data type (ADT). Each Ψ-type is equipped with a set of Ψ-*structures* which play an important role in our translation, enabling it to, e.g., efficiently combine subcomputations that produce ADTs with different privacy policies. Our experimental evaluation demonstrates that this strategy yields considerable performance improvements over the enforcement strategy used by Taype, yielding exponential improvements on the most complex benchmarks in our evaluation suite.

To summarize, the contributions of this chapter are as follows:

- We present Taypsi, a version of Taype extended with Ψ-types, a form of dependent sums that enables modular translation of non-secure programs into efficient, secure versions. This language is equipped with a security type system that offers the same

guarantees as Taype: after jointly computing a well-typed function, neither party can learn more about the other's private data than what can be gleaned from their own data and the output of the function.

- We develop an algorithm that combines a program written in the standard fragment of Taypsi with a privacy policy to produce a secure version that statically enforces the desired policy. We prove that this algorithm generates well-typed (and hence secure) target programs that are guaranteed to preserve the semantics of the source programs.
- We evaluate our approach on a range of case studies and microbenchmarks. Our experimental results demonstrate exponential performance improvements over the previous state-of-the-art (Taype) on several complicated benchmarks, while simultaneously showing no performance regression on the remaining benchmarks.

An artifact containing the Coq mechanization of the core calculus $\lambda_{\text{OADT}^\Psi}$, the implementation of Taypsi, its source code, and the source for all the benchmarks in our experiments with instructions is publicly available [60].

6.1 Overview

Before presenting the full details of our approach, we begin with an overview of Taypsi's strategy for building privacy-preserving applications. Consider the simple `filter` function in Figure 6.1, which drops all the elements in a list above a certain bound.[8] Suppose Alice owns some integers, and wants to know which of those integers are less than some threshold integer belonging to Bob, but neither party wants to share their data with the other. Using oblivious algebraic data types (OADTs) and the oblivious language from

```
data list = Nil | Cons ℤ list

fn filter : list → ℤ → list =
  λxs y ⇒
    match xs with
    | Nil ⇒ Nil
    | Cons x xs′ ⇒
      if x ≤ y
      then Cons x (filter xs′ y)
      else filter xs′ y
```

Figure 6.1. Filtering a list

[8] Taypsi supports higher-order functions, but our overview will use this specialized version for presentation purposes.

Chapter 3, we can encode a secure version of this `filter` function, allowing Alice and Bob to encrypt their data and then jointly compute `filter`, without leaking information about the encrypted data beyond what they can infer from the final disclosed output.

The particular policy (i.e., the chosen public view) that a secure application enforces can greatly impact the performance of that application, since the control flow of an application cannot depend on private data. In the case of our example, this means that the number of recursive calls to `filter` depends on the public information Alice is willing to share. If Alice only wants to share the maximum length of her list, for example, its encrypted version must be padded with dummy encrypted values, and a secure version of `filter` must recurse over these dummy elements, in order to avoid leaking information to Bob through its control flow. On the other hand, if Alice does not mind sharing the exact number of integers she owns, the joint computation will not have to go over these values, allowing a secure version of `filter` to be computed more efficiently.

Taypsi allows Alice and Bob to encode their private data and policies as OADTs. Figure 6.2 shows two OADTs for the type `list`: $\widehat{\texttt{list}}_{\leq}$, whose public view is the maximum length of a list, and $\widehat{\texttt{list}}_{=}$, whose public view is the exact length.

```
obliv list^≤ (k : ℕ) =
  if k = 0 then 𝟙
  else 𝟙 ⨥̂ ℤ^ × list^≤ (k−1)

obliv list^= (k : ℕ) =
  if k = 0 then 𝟙
  else ℤ^ × list^= (k−1)
```

Figure 6.2. Oblivious lists with maximum and exact length public views

Similar to Taype, in the implementation of Taypsi, oblivious values are represented using arrays of secure values. To ensure that attackers cannot learn anything from the "memory layout" of an OADT value, the size of this array is the same for all values of a particular OADT. As an example, the encoding of the list `Cons 10 (Cons 20 Nil)` as an oblivious list of type $\widehat{\texttt{list}}_{\leq}$ `2` is $\widehat{\texttt{inr}}$ `([10],` $\widehat{\texttt{inr}}$ `([20], ()))`, where $\widehat{\texttt{inr}}$ ($\widehat{\texttt{inl}}$) is the oblivious counterpart of standard sum injection `inr` (`inl`). Under the hood, this oblivious value is represented as an array holding four secure values; in the remainder of this section, we will informally write this value as `[Cons,10,Cons,20]`, where `[Cons]` is a synonym of the tag `[inr]` for readability. As another example, the empty list `Nil` is encoded as $\widehat{\texttt{inl}}$ `()`; it is also represented using an

array with four elements, `[Nil,−,−,−]`, where the last three elements are dummy encrypted values (denoted by −). Our compiler uses the type of $\widehat{\texttt{inl}}$ to automatically pad this array with these values, in order to ensure that it is indistinguishable from other private values of $\widehat{\texttt{list}_{\leq}}$ 2.

Users can directly implement privacy-preserving applications in Taypsi using OADTs and secure operations, but this requires manually instrumenting programs so that their control flow only depends on public information. Under this discipline, the implementation of a secure function intertwines program logic and privacy policies: the secure version of `filter` requires a different implementation depending on whether Alice is willing to share the exact length of her list, or an upper bound on that length. Taype (Chapter 5) decouples these concerns by allowing programs to include unsafe computations and repairing unsafe computations at runtime, using a novel form of semantics called *tape semantics*. As an example of this approach, in Taype, a secure implementation of `filter` that allows Alice to only share an upper bound on the size of her list can be written as:

```
fn filter≤ : (k : ℕ) → list≤ k → ℤ̂ → list≤ k =
  λk xs ŷ ⇒ list≤#s k (filter (list≤#r k xs) (ℤ̂#r ŷ))
```

The type signature of $\widehat{\texttt{filter}_{\leq}}$ specifies the policy it must follow. Intuitively, its implementation first "decrypts" the private inputs, applying the standard `filter` function to those values, and then "re-encrypts" the filtered list. In this example, the retractions of the private inputs $\widehat{\texttt{xs}}$ and $\hat{\texttt{y}}$ are unsafe computations that would violate the desired policy if they were computed naively. Fortunately, using the tape semantics prevents this from occurring by deferring these computations until it is safe to do so. Less fortunately, the runtime overhead of dynamic policy enforcement makes it hard to scale private applications manipulating structured data. As one data point, the secure version of `filter` produced by Taype takes more than 5 seconds to run with an oblivious list $\widehat{\texttt{list}_{\leq}}$ with sixteen elements, and its performance grows exponentially worse as the number of elements increases.

To understand the source of this slowdown, consider a computation that filters a private list containing 10 and 20 with integer 15: $\widehat{\texttt{filter}_{\leq}}$ `2 [Cons,10,Cons,20] [15]`. The first step in evaluating this function is to compute $\widehat{\texttt{list}_{\leq}}$`#r 2 [Cons,10,Cons,20]`. Completely

reducing this expression leaks information, so tape semantics instead stops evaluation after producing the following computation:

```
mux [false] Nil (Cons (ℤ̂#r [10]) (mux [false] Nil (Cons (ℤ̂#r [20]) Nil)))
```

The two `[false]`s are the results of securely checking if the two constructors in the input list are `Nil`. Observe that evaluating either `mux` or $\widehat{\mathbb{Z}}$`#r` would reveal private information, so the evaluation of these operations is deferred. This delayed computation can be thought of as an "if-tree" whose internal nodes are the private conditions needed to compute the final results, and whose leaves hold the result of the computation along each corresponding control flow path. To make progress, tape semantics distributes the context surrounding a delayed computation, `filter` and then $\widehat{\texttt{list}_{\leq}}$`#s` in this example, into each of its leaves; having done so, those leaves can be further evaluated. Importantly, in our example, the leaves of this if-tree are eventually re-encrypted using $\widehat{\texttt{list}_{\leq}}$`#s`. The tape semantics does so in a secure way, so that $\widehat{\mathbb{Z}}$`#r [10]` becomes `[10]` again, and each result list is converted to a secure value of the expected OADT. Once the branches of a `mux` node have been reduced to oblivious values of the same type, the node itself can be securely reduced using the secure semantics of `mux`. Unfortunately, the if-tree produced by the tape semantics can grow exponentially large before its `mux` nodes can be reduced. For example, after applying `filter` to the if-tree produced by $\widehat{\texttt{list}_{\leq}}$`#r`, the resulting if-tree has a leaf corresponding to every possible list that `filter` could produce; the number of these leaves is exponential in the maximum length of the input list. As any surrounding computation, i.e., $\widehat{\texttt{list}_{\leq}}$`#s` in our example, can be distributed to each of these leaves, an exponential number of computations may need to be performed before the if-tree can be collapsed.

To remedy these limitations, this chapter proposes to instead compile an insecure program into a secure version that *statically* enforces a specified policy. To do so, we extend Taype, the secure language from Chapter 5 with Ψ-*types*, a form of *dependent sums* (or dependent pairs) that packs public views and the oblivious data into a uniform representation. For example, $\Psi\widehat{\texttt{list}_{\leq}}$ is the oblivious list $\widehat{\texttt{list}_{\leq}}$ with its public view: $\langle$`2, inr ([10], inr ([20], ()))`$\rangle$ and $\langle$`2, inl ()`$\rangle$ are elements of type $\Psi\widehat{\texttt{list}_{\leq}}$, corresponding to the examples in the previous section. The first component of this pair-like syntax is a public view and the second component

is an OADT whose public view is exactly the first component. This allows users to again derive a private filter function from its type signature:

```
fn filter≤ : Ψlist≤ → ℤ → Ψlist≤ = %lift filter
```

Users no longer need to explicitly provide the public views for either the output or any intermediate subroutines: both are automatically inferred. As a result, the policy specification of $\widehat{\texttt{filter}}_{\leq}$ more directly corresponds to the type signature of `filter`. In addition, specifying policies using Ψ-types avoids mistakes in the supplied public views: using Taype, if the programmer mistakenly specifies the return type $\widehat{\texttt{list}}_{\leq}$ `(k−1)` for a secure version of `filter`, for example, the resulting implementation may truncate the last element of the result list. A keyword `%lift` is used to translate the standard non-secure function `filter` to a private version that respects the policy specification.

To understand how this translation works, consider a naive approach where each algebraic data type (ADT) is thought of as an abstract interface, whose operations correspond to the *introduction* and *elimination* forms of the algebraic data type[9]. An ADT, e.g., `list`, as well as any corresponding Ψ-type, e.g., $\Psi\widehat{\texttt{list}}_{\leq}$ and $\Psi\widehat{\texttt{list}}_{=}$, are implementations or *instances* of this interface. For example, an interface for list operations is:

```
ListLike t = {
  Nil : 𝟙 → t;
  Cons : ℤ × t → t;
  match : t → (𝟙 → α) → (ℤ × t → α) → α
}
```

As long as $\Psi\widehat{\texttt{list}}_{\leq}$ and $\Psi\widehat{\texttt{list}}_{=}$ implement this interface, we could straightforwardly translate `filter` to a secure version, as shown in Figure 6.3. This strategy does not rely on unsafe retractions like $\widehat{\texttt{list}}_{\leq}$`#r`, as private data always remains in its secure form, eliminating the need to defer unsafe computations, which is the source of exponential slowdowns in Taype. Unfortunately, there are several obstacles to directly implementing this strategy. First, an ADT and an OADT may not agree on the type signatures of the abstract interface. `ListLike` fixes the argument types of operations like `Cons` and `match`, meaning that `list` is

[9]↑As Taypsi already supports general recursion, we use pattern matching instead of recursion schemes as our elimination forms.

```
fn filter≤ : Ψlist≤ → ℤ̂ → Ψlist≤ = λxs y ⇒
  list≤#match xs
    (λ_ ⇒ list≤#Nil ())
    (λ(x, xs′) ⇒
      mux (x ≤̂ y) (list≤#Cons x (filter≤ xs′ y))
                  (filter≤ xs′ y))
```

(a) Secure `filter` using OADT $\widehat{\texttt{list}}_{\leq}$

```
fn filter= : Ψlist= → ℤ̂ → Ψlist= = λxs y ⇒
  list=#match xs
    (λ_ ⇒ list=#Nil ())
    (λ(x, xs′) ⇒
      mux (x ≤̂ y) (list=#Cons x (filter= xs′ y))
                  (filter= xs′ y))
```

(b) Secure `filter` using OADT $\widehat{\texttt{list}}_{=}$

Figure 6.3. Naive translation of `filter` to secure versions

not an instance of this abstract interface, despite `list` being a very reasonable (albeit very permissive) policy! In general, different OADTs may only be able to implement operations with specific signatures. Second, a private function may involve a mixture of oblivious types. Thus, some functions may need to coerce from one type to a "more" secure version. For example, if the policy of $\widehat{\texttt{filter}}_{\leq}$ is $\Psi\widehat{\texttt{list}}_{\leq} \to \mathbb{Z} \to \Psi\widehat{\texttt{list}}_{\leq}$, its second argument `y` will need to be converted to $\hat{\mathbb{Z}}$ in order to evaluate `x` $\hat{\leq}$ `y`. A secure list that discloses its exact length may similarly need to be converted to one disclosing its maximum length. Third, this naive translation results in ill-typed programs, because the branches of a `mux` may have mismatched public views. In $\widehat{\texttt{filter}}_{\leq}$, for example, the branches of `mux` may evaluate to ⟨`2, [Cons,10,Cons,20]`⟩ and ⟨`1, [Cons,20]`⟩, respectively. Thus, Taypsi's secure type system will (rightly) reject $\widehat{\texttt{filter}}_{\leq}$ as leaky. Lastly, the signatures that should be ascribed to any subsidiary function calls may not be obvious. Consider the following client of `filter`:

```
fn filter5 : list → list = λxs ⇒ filter xs 5
```

If `filter5` is given a signature $\Psi\widehat{\texttt{list}}_{\leq} \to \Psi\widehat{\texttt{list}}_{\leq}$, we would like to use a secure version of the `filter` function with the type $\Psi\widehat{\texttt{list}}_{\leq} \to \mathbb{Z} \to \Psi\widehat{\texttt{list}}_{\leq}$, as the threshold argument is publicly known. In general, a function may have many private versions, and we should infer which version to use at each callsite: a recursive function may even recursively call a different "version" of itself.

To solve these challenges, we generalize the abstract interface described above into a set of more flexible structures, which we collectively refer to as Ψ-structures (Section 6.3). Intuitively, each category of Ψ-structures solves one of the challenges described above. Our translation algorithm (Section 6.4) generates a set of typing *constraints* for the intermediate expressions in a program. These constraints are then solved using the set of available Ψ-structures, resulting in multiple private versions of the necessary functions and ruling out the infeasible ones, e.g., $\widehat{\texttt{filter}}_{=}$.

Figure 6.4 presents the methods of each category of Ψ-structures of $\widehat{\texttt{list}}_{\leq}$. The first two methods, $\widehat{\texttt{list}}_{\leq}$`#s` and $\widehat{\texttt{list}}_{\leq}$`#r`, are its section and retraction functions, belonging to the OADT-structure category. Unlike Taype, these two functions are not directly used to derive secure implementations of functions. In fact, our type system guarantees that retraction functions are never used in a secure computation, because Taypsi does not rely on tape semantics to repair unsafe computation (the `unsafe fn` keyword tells our type checker that $\widehat{\texttt{list}}_{\leq}$`#r` is potentially leaky). Our implementation of Taypsi exposes section and retraction functions as part of the API of the secure library it generates, however, so that client programs can conceal their private input and reveal the output of secure computations. This structure also includes a `view` method, which our translation uses to select the public view needed to safely convert a `list` into a $\Psi\widehat{\texttt{list}}_{\leq}$. Figure 6.4 does not show coercion methods, but the programmers can define a coercion from $\Psi\widehat{\texttt{list}}_{=}$ to $\Psi\widehat{\texttt{list}}_{\leq}$, for example.

The next set of methods belong to the intro-structure and elim-structure category. These introduction ($\widehat{\texttt{list}}_{\leq}$`#Nil` and $\widehat{\texttt{list}}_{\leq}$`#Cons`) and elimination ($\widehat{\texttt{list}}_{\leq}$`#match`) methods construct and destruct private list, respectively. As we construct and manipulate data, these methods build the private version, calculate its public view, and record that view in Ψ-types. Their type signatures are specified by the programmers, as long as the signatures are *compatible* with $\mathbb{Z} \times$ `list` (Section 6.3).

OADT-structure

```
fn list̂≤#s : (k : ℕ) → list → list̂≤ k = λk xs ⇒
  if k = 0 then ()
  else match xs with
       | Nil ⇒ inl̂ ()
       | Cons x xs′ ⇒ inr̂ (ℤ̂#s x, list̂≤#s (k−1) xs′)

unsafe fn list̂≤#r : (k : ℕ) → list̂≤ k → list = λk ⇒
  if k = 0 then λ_ ⇒ Nil
  else λxs ⇒ matcĥ xs with
              | inl̂ _ ⇒ Nil
              | inr̂ (x, xs′) ⇒ Cons (ℤ̂#r x) (list̂≤#r (k−1) xs′)

fn list̂≤#view : list → ℕ = length
```

Intro/elim-structure

```
fn list̂≤#Nil : 𝟙 → Ψlist̂≤ = λ_ ⇒ ⟨0, ()⟩

fn list̂≤#Cons : ℤ̂ × Ψlist̂≤ → Ψlist̂≤ = λ(x, ⟨k, xs⟩) ⇒ ⟨k+1, inr̂ (x, xs)⟩

fn list̂≤#match : Ψlist̂≤ → (𝟙 → α) → (ℤ̂ × Ψlist̂≤ → α) → α =
  λ⟨k, xs⟩ f1 f2 ⇒
    (if k = 0 then λ_ ⇒ f1 ()
     else λxs ⇒ matcĥ xs with
                | inl̂ _ ⇒ f1 ()
                | inr̂ (x, xs′) ⇒ f2 (x, ⟨k−1, xs′⟩) : list̂≤ k → α) xs
```

Join-structure

```
fn list̂≤#join : ℕ → ℕ → ℕ = max

fn list̂≤#reshape : (k : ℕ) → (k′ : ℕ) → list̂≤ k → list̂≤ k′ = λk k′ ⇒
  if k′ = 0 then λ_ ⇒ ()
  else if k = 0 then λ_ ⇒ inl̂ ()
       else λxs ⇒ matcĥ xs with
                  | inl̂ _ ⇒ inl̂ ()
                  | inr̂ (x, xs′) ⇒ inr̂ (x, list̂≤#reshape (k−1) (k′−1) xs′)
```

Figure 6.4. Ψ-structures of $\widehat{\texttt{list}}_{\leq}$

The `join` and `reshape` methods in the join-structure category enable translated programs to include private conditionals whose branches return OADT values with different public views. As an example, consider the following private conditional whose branches have Ψ-types:

`mux [true] ⟨2, [Cons,10,Cons,20]⟩ ⟨1, [Cons,20]⟩`

To build a version of this program that does not reveal `[true]`, Taypsi uses `join` to calculate a common public view that "covers" both branches. In this example, $\widehat{\texttt{list}}_{\leq}$`#join` chooses a public view of 2, as a list with at most one element also has at most two elements. Our translation then uses the `reshape` method to convert both branches to use this common public view. In our example, `[Cons,20]`, an oblivious list of maximum length 1, is converted into the list `[Cons,20,Nil,−]`, which has maximum length 2. Since both branches in the resulting program have the same public view, it is safe to evaluate `mux`: the resulting list is equivalent to `⟨2, mux [true] [Cons,10,Cons,20] [Cons,20,Nil,−]⟩`. As we will see later, not all OADTs admit join structures, e.g., $\widehat{\texttt{list}}_{=}$, but our translation generates constraints that take advantage of any that are available, failing when these constraints cannot be resolved in a way that guarantees security. Note that these two methods are key to avoiding the slowdown exhibited by Taype's enforcement strategy: they allow functions that may return different private representations to be *eagerly* evaluated, instead of being lazily deferred in a way that requires an exponential number of subcomputations to resolve.

In summary, to develop a secure application in Taypsi, programmers first implement its desired functionality, e.g., `filter`, in the public fragment of Taypsi, independently of any particular privacy policy. Policies are separately defined as oblivious algebraic data types, e.g., $\widehat{\texttt{list}}_{\leq}$, and their Ψ-structures. Users can then automatically derive a secure version of their application by providing the desired policy in the form of a type signature involving Ψ-types, relying on Taypsi's compiler to produce a privacy-preserving implementation. The type system of Taypsi, like Taype's, provides a strong security guarantee in the form of an obliviousness theorem (Theorem 6.2.1). This obliviousness theorem is a variant of noninterference [21], and ensures that well-typed programs in Taypsi are *secure by construction*: no private information can be inferred even by an attacker capable of observing every state of a program's execution. Our compilation algorithm is further guaranteed

to generate a secure implementation that preserves the behavior of the original program (Theorem 6.3.3). Taypsi's formal guarantees (Section 6.3.7) do not cover equi-termination of the source and target programs: when the public view lacks sufficient information to bound the computation of the original program, the secure version will not terminate, in order to avoid leaking information through its termination behavior.

The following three sections formally develop the language Taypsi, the Ψ-structures, and our translation algorithm.

6.2 Taypsi, Formally

This section presents $\lambda_{\texttt{OADT}\Psi}$, the core calculus for secure computation that we will use to explain our translation. This calculus extends the existing $\lambda_{\texttt{OADT}}$ calculus (Chapter 3) with Ψ-types, and uses ML-style ADTs in lieu of explicit `fold` and `unfold` operations. For simplicity, $\lambda_{\texttt{OADT}\Psi}$ does not include public sums and oblivious integers, which are straightforward to add.

6.2.1 Syntax

Figure 6.5 presents the syntax of $\lambda_{\texttt{OADT}\Psi}$. Types and expressions are in the same syntax class, as $\lambda_{\texttt{OADT}\Psi}$ is dependently typed, but we use `e` for expressions and τ for types when possible. A $\lambda_{\texttt{OADT}\Psi}$ program consists of a set of *global definitions* of data types, functions and oblivious types. Definitions in each of these classes are allowed to refer to themselves, permitting recursive types and general recursion in both function and oblivious type definitions. We use `x` for variable names, `C` for constructor names, `T` for type names, and $\widehat{\texttt{T}}$ for oblivious type names. Each constructor of an ADT definition takes exactly one argument, but this does not harm expressivity: this argument is $\mathbb{1}$ for constructors that take no arguments, e.g., `Nil`, and a tuple of types for constructors that have more than one argument, e.g., `Cons` takes an argument of type $\mathbb{Z} \times \texttt{list}$.

In addition to standard types and dependent function types (Π), $\lambda_{\texttt{OADT}\Psi}$ includes oblivious booleans ($\widehat{\mathbb{B}}$) and oblivious sum types ($\widehat{+}$). The elimination forms of these types are oblivious conditionals `mux` and oblivious case analysis $\widehat{\texttt{match}}$, respectively. The branches of both expressions must be private and each branch has to be fully evaluated before the expression

e, τ	::=		Expressions:
		$\mathbb{1} \mid \mathbb{B} \mid \widehat{\mathbb{B}} \mid \tau \times \tau \mid \tau \hat{+} \tau$	simple types
		Πx:τ,τ	dependent function type
		$\Psi\widehat{\text{T}}$	Ψ-type
		x | T	variable and type names
		() | b	unit and boolean constants
		λx:τ⇒e	function abstraction
		let x = e in e	let binding
		e e | C e | $\widehat{\text{T}}$ e	applications
		if e then e else e	conditional
		mux e e e	oblivious conditional
		(e,e)	pair
		⟨e,e⟩	dependent pair (Ψ-pair)
		π_b e	product and Ψ-type projection
		$\widehat{\iota_b}$<τ> e	oblivious sum injection
		$\widehat{\text{match e with x⇒e\|x⇒e}}$	oblivious sum elimination
		match e with $\overline{\text{C x⇒e}}$	ADT elimination
		$\widehat{\mathbb{B}}$#s e	boolean section
		[b] | [ι_b<$\widehat{\omega}$> $\hat{\text{v}}$]	runtime boxed values
D	::=		Global Definitions
		data T = $\overline{\text{C τ}}$	algebraic data type definition
		fn x:τ = e	(recursive) function definition
		obliv $\widehat{\text{T}}$ (x:τ) = τ	(recursive) oblivious type definition
$\widehat{\omega}$	::=	$\mathbb{1} \mid \widehat{\mathbb{B}} \mid \widehat{\omega} \times \widehat{\omega} \mid \widehat{\omega} \hat{+} \widehat{\omega}$	Oblivious Type Values
$\hat{\text{v}}$	::=	() | [b] | ($\hat{\text{v}}$,$\hat{\text{v}}$) | [ι_b<$\widehat{\omega}$> $\hat{\text{v}}$]	Oblivious Values
v	::=	$\hat{\text{v}}$ | b | (v,v) | ⟨v,v⟩ | λx:τ⇒e | C v	Values

Figure 6.5. $\lambda_{\text{OADT}^\Psi}$ syntax with extensions to λ_{OADT} highlighted

can take an atomic step to a final result. Boolean section $\widehat{\mathbb{B}}$#s is a primitive operation that "encrypts" a boolean expression to an oblivious version. Oblivious injection $\widehat{\iota_b}$ (i.e., $\widehat{\text{inl}}$ and $\widehat{\text{inr}}$) are the oblivious counterparts of the standard constructors for sums. Other terms are mostly standard, although let bindings (let), conditionals (if) and pattern matching (match) are allowed to return a type, as $\lambda_{\text{OADT}^\Psi}$ supports type-level computation.

The key addition over λ_{OADT} is the Ψ-*type*, $\Psi\widehat{\text{T}}$. It is constructed from a pair expression $\langle \cdot, \cdot \rangle$ that packs the public view and the oblivious data together, and has the same eliminators π_1

and π_2 as normal products. As an example, $\langle$`3, `$\widehat{\texttt{list}_{\leq}}$`#s 3 (Cons 1 Nil)`$\rangle$ creates a Ψ-pair of type $\Psi\widehat{\texttt{list}_{\leq}}$ with public view 3, using the section function from Figure 6.4. Projecting out the second component of a pair using π_2 produces a value of type $\widehat{\texttt{list}_{\leq}}$ `3`. A Ψ-type is essentially a dependent sum type (Σ`x:`τ`,`$\widehat{\texttt{T}}$ `x`), with the restriction that τ is the public view of $\widehat{\texttt{T}}$, and that $\widehat{\texttt{T}}$ `x` is an oblivious type.

Since $\lambda_{\texttt{OADT}\Psi}$ has type-level computation, oblivious types have normal forms; oblivious type values ($\widehat{\omega}$) are essentially polynomials formed by primitive oblivious types. We also have the oblivious values of oblivious boolean and sum type. Note that these "boxed" values only appear at runtime, our semantics use these to model encrypted booleans and tagged sums.

6.2.2 Semantics

Figure 6.6 shows a selection of the small-step semantics rules of $\lambda_{\texttt{OADT}\Psi}$ (the omitted rules are identical to $\lambda_{\texttt{OADT}}$), with judgment $\Sigma \vdash \texttt{e} \longrightarrow \texttt{e}'$. The global context Σ is a map from names to a global definition, which is elided for brevity as it is fixed in these rules. The semantics of $\lambda_{\texttt{OADT}\Psi}$ is similar to $\lambda_{\texttt{OADT}}$, with the addition of S-PsiProj to handle the projection of dependent pairs, which is simply the same as normal projection. S-Ctx reduces subterms according to the evaluation contexts defined in Figure 6.6. The first few contexts take care of the type-level reduction of product and oblivious sum type. The type annotation of oblivious injection ι_b is reduced to a type value first, before reducing the payload. The evaluation contexts for `mux` capture the intuition that all components of a private conditional have to be normalized to values first to avoid leaking the private condition through control flow channels.

S-OMatch evaluates a pattern matching expression for oblivious sums. Similar to `mux`, oblivious pattern matching needs to ensure the reduction does not reveal private information about the discriminee, e.g., whether it is the left injection or right injection. To do so, we reduce a $\widehat{\texttt{match}}$ to an oblivious conditional that uses the private tag. The pattern variable in the "correct" branch is of course instantiated by the payload in the discriminee, while the pattern variable in the "wrong" branch is an arbitrary value of the corresponding type, synthesized

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

$$\frac{\texttt{e} \longrightarrow \texttt{e}'}{\mathcal{E}[\texttt{e}] \longrightarrow \mathcal{E}[\texttt{e}']}\ \textsc{S-Ctx} \qquad \frac{\texttt{fn x:}\tau\texttt{ = e} \in \Sigma}{\texttt{x} \longrightarrow \texttt{e}}\ \textsc{S-Fun} \qquad \frac{\texttt{obliv }\widehat{\texttt{T}}\texttt{ (x:}\tau\texttt{) = }\tau' \in \Sigma}{\widehat{\texttt{T}}\texttt{ v} \longrightarrow [\texttt{v/x}]\tau'}\ \textsc{S-OADT}$$

$$\frac{}{(\lambda\texttt{x:}\tau\Rightarrow\texttt{e})\texttt{ v} \longrightarrow [\texttt{v/x}]\texttt{e}}\ \textsc{S-App} \qquad \frac{}{\texttt{if b then e}_1\texttt{ else e}_2 \longrightarrow \texttt{ite(b,e}_1\texttt{,e}_2\texttt{)}}\ \textsc{S-If}$$

$$\frac{}{\texttt{mux [b] v}_1\texttt{ v}_2 \longrightarrow \texttt{ite(b,v}_1\texttt{,v}_2\texttt{)}}\ \textsc{S-Mux} \qquad \frac{}{\texttt{match C}_i\texttt{ v with }\overline{\texttt{C x}\Rightarrow\texttt{e}} \longrightarrow [\texttt{v/x}]\texttt{e}_i}\ \textsc{S-Match}$$

$$\frac{}{\pi_b\ (\texttt{v}_1\texttt{,v}_2) \longrightarrow \texttt{ite(b,v}_1\texttt{,v}_2\texttt{)}}\ \textsc{S-Proj} \qquad \frac{}{\widehat{\mathbb{B}}\texttt{\#s b} \longrightarrow \texttt{[b]}}\ \textsc{S-Sec} \qquad \frac{}{\widehat{\iota}_b\texttt{<}\widehat{\omega}\texttt{>}\ \widehat{\texttt{v}} \longrightarrow [\iota_b\texttt{<}\widehat{\omega}\texttt{>}\ \widehat{\texttt{v}}]}\ \textsc{S-OInj}$$

$$\frac{}{\pi_b\ \langle\texttt{v}_1\texttt{,v}_2\rangle \longrightarrow \texttt{ite(b,v}_1\texttt{,v}_2\texttt{)}}\ \textsc{S-PsiProj}$$

$$\frac{\widehat{\texttt{v}}_1 \Leftarrow \widehat{\omega}_1 \qquad \widehat{\texttt{v}}_2 \Leftarrow \widehat{\omega}_2}{\widehat{\texttt{match}}\ [\iota_b\texttt{<}\widehat{\omega}_1\widehat{+}\widehat{\omega}_2\texttt{>}\ \widehat{\texttt{v}}]\ \texttt{with x}\Rightarrow\texttt{e}_1\texttt{|x}\Rightarrow\texttt{e}_2 \longrightarrow \begin{array}{l}\texttt{mux [b] ite(b,}[\widehat{\texttt{v}}/\texttt{x}]\texttt{e}_1\texttt{,}[\widehat{\texttt{v}}_1/\texttt{x}]\texttt{e}_1\texttt{)} \\ \qquad\qquad \texttt{ite(b,}[\widehat{\texttt{v}}_2/\texttt{x}]\texttt{e}_2\texttt{,}[\widehat{\texttt{v}}/\texttt{x}]\texttt{e}_2\texttt{)}\end{array}}\ \textsc{S-OMatch}$$

Evaluation Contexts

$$\begin{array}{rcl}\mathcal{E} & ::= & \square\times\tau \mid \widehat{\omega}\times\square \mid \square\widehat{+}\tau \mid \widehat{\omega}\widehat{+}\square \\ & \mid & \texttt{let x = }\square\texttt{ in e} \mid \texttt{e}\ \square \mid \square\ \texttt{v} \mid \texttt{C}\ \square \mid \widehat{\texttt{T}}\ \square \\ & \mid & \texttt{if }\square\texttt{ then e else e} \mid \texttt{mux}\ \square\ \texttt{e e} \mid \texttt{mux v}\ \square\ \texttt{e} \mid \texttt{mux v v}\ \square \\ & \mid & \texttt{(}\square\texttt{,e)} \mid \texttt{(v,}\square\texttt{)} \mid \langle\square\texttt{,e}\rangle \mid \langle\texttt{v,}\square\rangle \mid \pi_b\ \square \\ & \mid & \widehat{\iota}_b\texttt{<}\square\texttt{>}\ \texttt{e} \mid \widehat{\iota}_b\texttt{<}\widehat{\omega}\texttt{>}\ \square \mid \widehat{\texttt{match}}\ \square\ \texttt{with x}\Rightarrow\texttt{e|x}\Rightarrow\texttt{e} \\ & \mid & \texttt{match}\ \square\ \texttt{with}\ \overline{\texttt{C x}\ \Rightarrow\ \texttt{e}} \mid \widehat{\mathbb{B}}\texttt{\#s}\ \square\end{array}$$

Figure 6.6. Selected small-step semantics rules of $\lambda_{\texttt{OADT}\Psi}$

from the judgment $\widehat{\texttt{v}} \Leftarrow \widehat{\omega}$, whose definition has been presented in Figure 3.9. When evaluating a $\widehat{\texttt{match}}$ statement whose discriminee is $[\texttt{inl}\texttt{<}\widehat{\mathbb{B}}\widehat{+}\widehat{\mathbb{B}}\times\widehat{\mathbb{B}}\texttt{>}\ [\texttt{true}]]$, the pattern variable in the second branch can be substituted by $([\texttt{true}],[\texttt{true}])$, $([\texttt{false}],[\texttt{true}])$, or any other pair of oblivious booleans.

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

$$\textsc{T-Conv}\ \frac{\Gamma \vdash \mathtt{e} : \tau \qquad \tau \equiv \tau' \qquad \Gamma \vdash \tau' :: *}{\Gamma \vdash \mathtt{e} : \tau'} \qquad \textsc{T-Abs}\ \frac{\mathtt{x} : \tau_1, \Gamma \vdash \mathtt{e} : \tau_2 \qquad \Gamma \vdash \tau_1 :: *}{\Gamma \vdash \lambda \mathtt{x}{:}\tau_1 \Rightarrow \mathtt{e} : \Pi \mathtt{x}{:}\tau_1, \tau_2}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash \mathtt{e}_2 : \Pi \mathtt{x}{:}\tau_1, \tau_2 \qquad \Gamma \vdash \mathtt{e}_1 : \tau_1}{\Gamma \vdash \mathtt{e}_2\ \mathtt{e}_1 : [\mathtt{e}_1/\mathtt{x}]\tau_2} \qquad \textsc{T-If}\ \frac{\Gamma \vdash \mathtt{e}_0 : \mathbb{B} \qquad \Gamma \vdash \mathtt{e}_1 : [\mathtt{true}/\mathtt{z}]\tau \qquad \Gamma \vdash \mathtt{e}_2 : [\mathtt{false}/\mathtt{z}]\tau}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 : [\mathtt{e}_0/\mathtt{z}]\tau}$$

$$\textsc{T-Ctor}\ \frac{\mathtt{data}\ \mathtt{T} = \overline{\mathtt{C}\ \tau} \in \Sigma \qquad \Gamma \vdash \mathtt{e} : \tau_i}{\Gamma \vdash \mathtt{C}_i\ \mathtt{e} : \mathtt{T}} \qquad \textsc{T-Match}\ \frac{\mathtt{data}\ \mathtt{T} = \overline{\mathtt{C}\ \tau} \in \Sigma \qquad \Gamma \vdash \mathtt{e}_0 : \mathtt{T} \qquad \forall i.\ \mathtt{x} : \tau_i, \Gamma \vdash \mathtt{e}_i : [\mathtt{C}_i\ \mathtt{x}/\mathtt{z}]\tau'}{\Gamma \vdash \mathtt{match}\ \mathtt{e}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{e}} : [\mathtt{e}_0/\mathtt{z}]\tau'}$$

$$\textsc{T-Mux}\ \frac{\Gamma \vdash \mathtt{e}_0 : \widehat{\mathbb{B}} \qquad \Gamma \vdash \tau :: *^{\mathsf{O}} \qquad \Gamma \vdash \mathtt{e}_1 : \tau \qquad \Gamma \vdash \mathtt{e}_2 : \tau}{\Gamma \vdash \mathtt{mux}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2 : \tau} \qquad \textsc{T-PsiPair}\ \frac{\mathtt{obliv}\ \widehat{\mathtt{T}}\ (\mathtt{x}{:}\tau) = \tau' \in \Sigma \qquad \Gamma \vdash \mathtt{e}_1 : \tau \qquad \Gamma \vdash \mathtt{e}_2 : \widehat{\mathtt{T}}\ \mathtt{e}_1}{\Gamma \vdash \langle \mathtt{e}_1, \mathtt{e}_2 \rangle : \Psi \widehat{\mathtt{T}}}$$

$$\textsc{T-PsiProj}_1\ \frac{\mathtt{obliv}\ \widehat{\mathtt{T}}\ (\mathtt{x}{:}\tau) = \tau' \in \Sigma \qquad \Gamma \vdash \mathtt{e} : \Psi \widehat{\mathtt{T}}}{\Gamma \vdash \pi_1\ \mathtt{e} : \tau} \qquad \textsc{T-PsiProj}_2\ \frac{\mathtt{obliv}\ \widehat{\mathtt{T}}\ (\mathtt{x}{:}\tau) = \tau' \in \Sigma \qquad \Gamma \vdash \mathtt{e} : \Psi \widehat{\mathtt{T}}}{\Gamma \vdash \pi_2\ \mathtt{e} : \widehat{\mathtt{T}}\ (\pi_1\ \mathtt{e})}$$

Figure 6.7. Selected typing rules of $\lambda_{\mathtt{OADT}\Psi}$

6.2.3 Type System

Similar to $\lambda_{\mathtt{OADT}}$, types in $\lambda_{\mathtt{OADT}\Psi}$ are classified by *kinds* which specify how protected a type is, in addition to ensuring the types are well-formed. For example, an oblivious type, e.g., $\widehat{\mathbb{B}}$, kinded by $*^{\mathsf{O}}$, can be used as branches of an oblivious conditional, but not as a public view, which can only be kinded by $*^{\mathsf{P}}$. A mixed kind $*^{\mathsf{M}}$ is used to classify function types and types that consist of both public and oblivious components, e.g., $\mathbb{B} \times \widehat{\mathbb{B}}$. A type with a mixed kind cannot be used as a public view or in private context.

The type system of $\lambda_{\mathtt{OADT}\Psi}$ is defined by a pair of typing and kinding judgments, $\Sigma; \Gamma \vdash \mathtt{e} : \tau$ and $\Sigma; \Gamma \vdash \tau :: \kappa$, with global context Σ (which is again elided for brevity) and the standard

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

$$\text{K-Sub}\ \frac{\Gamma \vdash \tau :: \kappa \qquad \kappa \sqsubseteq \kappa'}{\Gamma \vdash \tau :: \kappa'} \qquad \text{K-OADT}\ \frac{\texttt{obliv}\ \widehat{\texttt{T}}\ (\texttt{x}{:}\tau)\ =\ \tau' \in \Sigma \qquad \Gamma \vdash \texttt{e} : \tau}{\Gamma \vdash \widehat{\texttt{T}}\ \texttt{e} :: *^{\texttt{O}}}$$

$$\text{K-Pi}\ \frac{\Gamma \vdash \tau_1 :: * \qquad \texttt{x} : \tau_1, \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \Pi \texttt{x}{:}\tau_1\texttt{,}\tau_2 :: *^{\texttt{M}}} \qquad \text{K-OSum}\ \frac{\Gamma \vdash \tau_1 :: *^{\texttt{O}} \qquad \Gamma \vdash \tau_2 :: *^{\texttt{O}}}{\Gamma \vdash \tau_1 \widehat{+} \tau_2 :: *^{\texttt{O}}}$$

$$\text{K-Psi}\ \frac{\texttt{obliv}\ \widehat{\texttt{T}}\ (\texttt{x}{:}\tau)\ =\ \tau' \in \Sigma}{\Gamma \vdash \Psi\widehat{\texttt{T}} :: *^{\texttt{M}}} \qquad \text{K-If}\ \frac{\Gamma \vdash \texttt{e}_0 : \mathbb{B} \qquad \Gamma \vdash \tau_1 :: *^{\texttt{O}} \qquad \Gamma \vdash \tau_2 :: *^{\texttt{O}}}{\Gamma \vdash \texttt{if}\ \texttt{e}_0\ \texttt{then}\ \tau_1\ \texttt{else}\ \tau_2 :: *^{\texttt{O}}}$$

$$\text{K-Match}\ \frac{\texttt{data}\ \texttt{T}\ =\ \overline{\texttt{C}\ \tau} \in \Sigma \qquad \Gamma \vdash \texttt{e}_0 : \texttt{T} \qquad \forall i.\ \texttt{x} : \tau_i, \Gamma \vdash \tau'_i :: *^{\texttt{O}}}{\Gamma \vdash \texttt{match}\ \texttt{e}_0\ \texttt{with}\ \overline{\texttt{C}\ \texttt{x} \Rightarrow \tau'} :: *^{\texttt{O}}}$$

Figure 6.8. Selected kinding rules of $\lambda_{\texttt{OADT}\Psi}$

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

$$\text{DT-Fun}\ \frac{\cdot \vdash \tau :: * \qquad \cdot \vdash \texttt{e} : \tau}{\Sigma \vdash \texttt{fn}\ \texttt{x}{:}\tau\ =\ \texttt{e}} \qquad \text{DT-ADT}\ \frac{\forall i.\ \cdot \vdash \tau_i :: *^{\texttt{P}}}{\Sigma \vdash \texttt{data}\ \texttt{T}\ =\ \overline{\texttt{C}\ \tau}} \qquad \text{DT-OADT}\ \frac{\cdot \vdash \tau :: *^{\texttt{P}} \qquad \texttt{x} : \tau \vdash \tau' :: *^{\texttt{O}}}{\Sigma \vdash \texttt{obliv}\ \widehat{\texttt{T}}\ (\texttt{x}{:}\tau)\ =\ \tau'}$$

Figure 6.9. $\lambda_{\texttt{OADT}\Psi}$ global definitions typing

typing context Γ. Figure 6.7 and Figure 6.8 presents a subset of our typing and kinding rules; the omitted rules are identical to the ones in $\lambda_{\texttt{OADT}}$.

The security type system [27] of $\lambda_{\texttt{OADT}\Psi}$ enforces a few key invariants. First, oblivious types can only be constructed from oblivious types, which is enforced by the kinding rules, such as K-OSum. Otherwise, the attacker could infer the private tag of an oblivious sum, e.g., $\mathbb{B}\widehat{+}\mathbb{1}$, by observing its public payload. Second, oblivious operations, e.g., `mux`, require their subterms to be oblivious, to avoid leaking private information via control flow channels. T-Mux, for example, requires both branches to be typed by an oblivious type, otherwise an attacker may infer the private condition by observing the result, as in `mux [true] true false`. Third,

type-level computation is only defined for oblivious types and cannot depend on private information. Thus, K-If and K-Match requires all branches to have oblivious kinds, and the condition to be public. The type `mux [true]` $\mathbb{1}$ $\widehat{\mathbb{B}}$ is ill-typed, since the "shape" of the data reveals the private condition.

The typing rules for Ψ-types are defined similarly to the rules of standard dependent sums. T-PsiPair introduces a dependent pair, where the type of the second component depends on the first component. In contrast to standard dependent sum type, Ψ-type has the restriction that the first component must be public, and the second component must be oblivious. This condition is implicitly enforced by the side condition that $\widehat{\mathsf{T}}$ is an OADT with public view type τ. Figure 6.9 shows the typing rules for global definitions; DT-OADT prescribes exactly this restriction. The rules for the first and second projection of Ψ-type, T-PsiProj$_1$ and T-PsiProj$_2$, are very similar to the corresponding rules for standard dependent sum types. Observe that a Ψ-type always has mixed kind, as in K-Psi, because it consists of both public and oblivious components.

T-Conv allows conversion between equivalent types, such as `if true then` $\widehat{\mathbb{B}}$ `else` $\mathbb{1}$ and $\widehat{\mathbb{B}}$. The equivalence judgment $\tau \equiv \tau'$ is defined by a set of *parallel reduction* rules, which are mostly identical to the rules in $\lambda_{\texttt{OADT}}$. The converted type is nonetheless required to be well-kinded.

Note that these rules cannot be used to type check retraction functions, e.g., $\widehat{\texttt{list}_{\leq}}$`#r` from Figure 6.4, and for good reason: these functions reveal private information. Nevertheless, we still want to check that these sorts of "leaky" functions have standard type safety properties, i.e., progress and preservation. To do so, we use a version of these rules that simply omit some security-related side-conditions about oblivious kinding: removing $\Gamma \vdash \tau :: *^{\mathsf{O}}$ from T-Mux allows the branches of a `mux` to disclose the private condition, for example. The implementation of Taypsi's type checker uses a "mode" flag to indicate whether security-related side-conditions should be checked. Our implementation ensures that secure functions never use any leaky functions.

6.2.4 Metatheory

With our addition of Ψ-types, $\lambda_{\texttt{OADT}\Psi}$ enjoys the standard type safety properties (i.e., progress and preservation), and, more importantly, the same security guarantees as $\lambda_{\texttt{OADT}}$:

Theorem 6.2.1 (Obliviousness). *If* $\mathsf{e}_1 \approx \mathsf{e}_2$ *and* $\cdot \vdash \mathsf{e}_1 : \tau_1$ *and* $\cdot \vdash \mathsf{e}_2 : \tau_2$*, then*

1. $\mathsf{e}_1 \longrightarrow^n \mathsf{e}_1'$ *if and only if* $\mathsf{e}_2 \longrightarrow^n \mathsf{e}_2'$ *for some* e_2'.

2. *if* $\mathsf{e}_1 \longrightarrow^n \mathsf{e}_1'$ *and* $\mathsf{e}_2 \longrightarrow^n \mathsf{e}_2'$ *, then* $\mathsf{e}_1' \approx \mathsf{e}_2'$.

Here, $\mathsf{e} \approx \mathsf{e}'$ means the two expressions are indistinguishable, i.e., they only differ in their unobservable oblivious values, and $\mathsf{e} \longrightarrow^n \mathsf{e}'$ means e reduces to e' in exactly n steps. This obliviousness theorem provides a strong security guarantee: well-typed programs that are indistinguishable produce traces that are pairwise indistinguishable. In other words, an attacker cannot infer any private information even by observing the execution trace of a program. All these results are mechanized in the Coq theorem prover, including the formalization of the core calculus and the proofs of soundness and obliviousness theorems.

6.3 Ψ-structures and Declarative Lifting

While our secure language makes it possible to encode structured data and privacy policies, and use them in a secure way, it does not quite achieve our main goal yet, i.e., to decouple privacy policies and programmatic concerns. To do so, we allow the programmers to implement the functionality of their secure application in a conventional way, that is using only the public, nondependent fragment of Taypsi. We make this fragment explicit by requiring such programs to have *simple types*, denoted by η, defined in Figure 6.10. For example, `filter` has simple type $\texttt{list} \to \mathbb{Z} \to \texttt{list}$. Programs of simple types are the source programs to our lifting process that translates them to a private version against a policy, which stipulates the public information allowed to disclose in the program input and output. This policy on private functionality is specified by a *specification type*, denoted by θ, defined also in Figure 6.10. For example, $\widehat{\texttt{filter}}_{\leq}$ has specification type $\Psi\widehat{\texttt{list}}_{\leq} \to \hat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}}_{\leq}$. Note that dependent types are not directly allowed in specifications, they are instead encapsulated

$\boxed{\lfloor\theta\rfloor}$

Simple types

$\eta ::= \mathbb{1} \mid \mathbb{B} \mid \mathtt{T} \mid \eta\times\eta \mid \eta\to\eta$

Specification types

$\theta ::= \mathbb{1} \mid \mathbb{B} \mid \widehat{\mathbb{B}} \mid \mathtt{T} \mid \Psi\widehat{\mathtt{T}} \mid \theta\times\theta \mid \theta\to\theta$

$\lfloor\mathbb{1}\rfloor = \mathbb{1} \qquad \lfloor\mathbb{B}\rfloor = \lfloor\widehat{\mathbb{B}}\rfloor = \mathbb{B}$

$\lfloor\mathtt{T}\rfloor = \mathtt{T}$ where $\mathtt{T}$ is an ADT

$\lfloor\Psi\widehat{\mathtt{T}}\rfloor = \mathtt{T}$ where $\widehat{\mathtt{T}}$ is an OADT for $\mathtt{T}$

$\lfloor\theta\times\theta\rfloor = \lfloor\theta\rfloor\times\lfloor\theta\rfloor \qquad \lfloor\theta\to\theta\rfloor = \lfloor\theta\rfloor\to\lfloor\theta\rfloor$

Figure 6.10. Simple types, specification types and erasure

in Ψ-types. Simple types and specification types are additionally required to be well-kinded under empty local context, i.e., all ADTs and OADTs appear in them are defined.

However, not all specification types are valid with respect to a simple type. It is nonsensical to give $\widehat{\texttt{filter}}$ the specification type $\widehat{\mathbb{Z}} \to \widehat{\mathbb{B}}$, for example. The specification types should still correspond to the simple types in some way: the specification type corresponding to `list` should at least be "list-like". This correspondence is formally captured in the erasure function in Figure 6.10, which maps a specification type to the "underlying" simple type. For example, $\Psi\widehat{\texttt{list}}_{\leq}$ is erased to `list`. This function clearly induces an equivalence relation: the erasure $\lfloor\theta\rfloor$ is the representative of the equivalence class. We call this equivalence class a *compatibility class*, and say two types are *compatible* if they belong to the same compatibility class. For example, `list`, $\Psi\widehat{\texttt{list}}_{\leq}$ and $\Psi\widehat{\texttt{list}}_{=}$ are in the same compatibility class `[list]`. This erasure operation is straightforwardly extended to typing contexts, $\lfloor\Gamma\rfloor$, by erasing every specification type in Γ and leaving other types untouched.

Our translation transforms source programs with simple types into target programs with the desired (compatible) specification types. As mentioned in Section 6.1, this *lifting* process depends on a set of Ψ-*structures* which explain how to translate certain operations associated with an OADT.

6.3.1 OADT Structures

Every global OADT definition $\widehat{\mathtt{T}}$ must be equipped with an *OADT-structure*, defined below.

Definition 6.3.1 (OADT-structure)**.** An OADT-structure of an OADT $\widehat{\mathtt{T}}$, with public view type τ, consists of the following (Taypsi) type and functions:

- A public type $\mathtt{T} :: *^{\mathsf{P}}$, which is the public counterpart of $\widehat{\mathtt{T}}$. We say $\widehat{\mathtt{T}}$ is an OADT for $\mathtt{T}$.
- A section function $\mathtt{s} : \Pi \mathtt{k}{:}\tau, \mathtt{T} \to \widehat{\mathtt{T}}\ \mathtt{k}$, which converts a public type to its oblivious counterpart.
- A retraction function $\mathtt{r} : \Pi \mathtt{k}{:}\tau, \widehat{\mathtt{T}}\ \mathtt{k} \to \mathtt{T}$, which converts an oblivious type to its public version.
- A public view function $\nu : \mathtt{T} \to \tau$, which creates a valid view of the public type.
- A binary relation $\preccurlyeq$ over values of types $\mathtt{T}$ and τ; $\mathtt{v} \preccurlyeq \mathtt{k}$ reads as $\mathtt{v}$ has public view $\mathtt{k}$, or $\mathtt{k}$ is a valid public view of $\mathtt{v}$.

These operations are required to satisfy the following axioms:

- (A-O_1) $\mathtt{s}$ and $\mathtt{r}$ are a valid section and retraction, i.e., $\mathtt{r}$ is a left-inverse for $\mathtt{s}$, given a valid public view: for any values $\mathtt{v} : \mathtt{T}$, $\mathtt{k} : \tau$ and $\widehat{\mathtt{v}} : \widehat{\mathtt{T}}\ \mathtt{k}$, if $\mathtt{v} \preccurlyeq \mathtt{k}$ and $\mathtt{s}\ \mathtt{k}\ \mathtt{v} \longrightarrow^* \widehat{\mathtt{v}}$, then $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{v}$.
- (A-O_2) the result of $\mathtt{r}$ always has valid public view: $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{v}$ implies $\mathtt{v} \preccurlyeq \mathtt{k}$ for all values $\mathtt{k} : \tau$, $\widehat{\mathtt{v}} : \widehat{\mathtt{T}}\ \mathtt{k}$ and $\mathtt{v} : \mathtt{T}$.
- (A-O_3) ν produces a valid public view: $\nu\ \mathtt{v} \longrightarrow^* \mathtt{k}$ implies $\mathtt{v} \preccurlyeq \mathtt{k}$, given any values $\mathtt{v} : \mathtt{T}$ and $\mathtt{k} : \tau$.

For example, $\widehat{\texttt{list}_{\leq}}$ is equipped with the OADT-structure with the public type `list`, section function $\widehat{\texttt{list}_{\leq}}$`#s`, retraction function $\widehat{\texttt{list}_{\leq}}$`#r` and view function $\widehat{\texttt{list}_{\leq}}$`#view`, all of which are shown in Figure 6.4. Taypsi users do not need to explicitly give the public type of an OADT-structure, as it can be inferred from the types of the other functions. The binary relation $\preccurlyeq$ is only used in the proof of correctness of our translation, so Taypsi users can also elide it. In the case of $\widehat{\texttt{list}_{\leq}}$, $\preccurlyeq$ simply states the length of the list is no larger than the public view.

6.3.2 Join Structures

In order for Ψ-types to be flexibly used in the branches of secure control flow structures, our translation must be able to find a common public view for both branches, and to convert an OADT to use this view. To do so, an OADT can *optionally* be equipped with a *join-structure*.

Definition 6.3.2 (join-structure)**.** A join-structure of an OADT $\widehat{\mathtt{T}}$ for $\mathtt{T}$, with public view type τ, consists of the following operations:

- A binary relation $\sqsubseteq$ on τ, used to compare two public views.
- A join function $\sqcup : \tau \to \tau \to \tau$, which computes an upper bound of two public views[10].
- A reshape function $\upharpoonright : \Pi \mathtt{k}:\tau, \Pi \mathtt{k}':\tau, \widehat{\mathtt{T}}\ \mathtt{k} \to \widehat{\mathtt{T}}\ \mathtt{k}'$, which converts an OADT to one with a different public view.

such that:

- (A-R$_1$) $\sqsubseteq$ is a partial order on τ.
- (A-R$_2$) the join function produces an upper bound: given values $\mathtt{k}_1$, $\mathtt{k}_2$ and $\mathtt{k}$ of type τ, if $\mathtt{k}_1 \sqcup \mathtt{k}_2 \longrightarrow^* \mathtt{k}$, then $\mathtt{k}_1 \sqsubseteq \mathtt{k}$ and $\mathtt{k}_2 \sqsubseteq \mathtt{k}$.
- (A-R$_3$) the validity of public views is monotone with respect to the binary relation $\sqsubseteq$: for any values $\mathtt{v} : \mathtt{T}$, $\mathtt{k} : \tau$ and $\mathtt{k}' : \tau$, if $\mathtt{v} \preccurlyeq \mathtt{k}$ and $\mathtt{k} \sqsubseteq \mathtt{k}'$, then $\mathtt{v} \preccurlyeq \mathtt{k}'$.
- (A-R$_4$) the reshape function produces equivalent value, as long as the new public view is valid: for any values $\mathtt{v} : \mathtt{T}$, $\mathtt{k} : \tau$, $\mathtt{k}' : \tau$, $\widehat{\mathtt{v}} : \widehat{\mathtt{T}}\ \mathtt{k}$ and $\widehat{\mathtt{v}}' : \widehat{\mathtt{T}}\ \mathtt{k}'$, if $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{v}$ and $\mathtt{v} \preccurlyeq \mathtt{k}'$ and $\upharpoonright\ \mathtt{k}\ \mathtt{k}'\ \widehat{\mathtt{v}} \longrightarrow^* \widehat{\mathtt{v}}'$, then $\mathtt{r}\ \mathtt{k}'\ \widehat{\mathtt{v}}' \longrightarrow^* \mathtt{v}$.

Figure 6.4 defines the join and reshape functions $\widehat{\mathtt{list}_{\leq}}\mathtt{\#join}$ and $\widehat{\mathtt{list}_{\leq}}\mathtt{\#reshape}$. The partial order for this join structure is simply the total order on integers, and the join is simply the maximum of the two numbers. Not all OADTs have a sensible join-structure: oblivious lists using their exact length as a public view cannot be combined if they have different

[10]↑It is a bit misleading to call the operation $\sqcup$ "join", as it only computes an upper bound, not necessarily the lowest one. However, it *should* compute a supremum for performance reasons: intuitively, larger public view means more padding.

$$\boxed{\curlywedgeuparrow\theta \rhd \widehat{\texttt{ite}}}$$

$$\frac{\theta \in \{\mathbb{1}, \widehat{\mathbb{B}}\}}{\curlywedgeuparrow\theta \rhd \lambda\hat{\texttt{b}}\ \texttt{x}\ \texttt{y}{\Rightarrow}\texttt{mux}\ \hat{\texttt{b}}\ \texttt{x}\ \texttt{y}}$$

$$\frac{\curlywedgeuparrow\theta_1 \rhd \widehat{\texttt{ite}}_1 \qquad \curlywedgeuparrow\theta_2 \rhd \widehat{\texttt{ite}}_2}{\curlywedgeuparrow\theta_1{\times}\theta_2 \rhd \lambda\hat{\texttt{b}}\ \texttt{x}\ \texttt{y}{\Rightarrow}(\widehat{\texttt{ite}}_1\ \hat{\texttt{b}}\ (\pi_1\ \texttt{x})\ (\pi_1\ \texttt{y}), \widehat{\texttt{ite}}_2\ \hat{\texttt{b}}\ (\pi_2\ \texttt{x})\ (\pi_2\ \texttt{y}))}$$

$$\frac{\curlywedgeuparrow\theta_2 \rhd \widehat{\texttt{ite}}_2}{\curlywedgeuparrow\theta_1{\rightarrow}\theta_2 \rhd \lambda\hat{\texttt{b}}\ \texttt{x}\ \texttt{y}{\Rightarrow}\lambda\texttt{z}{\Rightarrow}\widehat{\texttt{ite}}_2\ \hat{\texttt{b}}\ (\texttt{x}\ \texttt{z})\ (\texttt{y}\ \texttt{z})}$$

$$\frac{(\hat{\texttt{T}}, \sqcup, \Uparrow) \in \mathcal{S}_\sqcup}{\curlywedgeuparrow\Psi\hat{\texttt{T}} \rhd \begin{array}{l}\lambda\hat{\texttt{b}}\ \texttt{x}\ \texttt{y}{\Rightarrow}\texttt{let}\ \texttt{k} = \pi_1\ \texttt{x} \sqcup \pi_1\ \texttt{y}\ \texttt{in} \\ \quad \langle \texttt{k}, \texttt{mux}\ \hat{\texttt{b}}\ (\Uparrow\ (\pi_1\ \texttt{x})\ \texttt{k}\ (\pi_2\ \texttt{x})) \\ \qquad (\Uparrow\ (\pi_1\ \texttt{y})\ \texttt{k}\ (\pi_2\ \texttt{y}))\rangle\end{array}}$$

Figure 6.11. Mergeability

lengths. If such lists are the branches of an oblivious conditional, lifting will either fail or coerce both to an OADT with a join-structure.

Join structures induce a *mergeability* relation, defined in Figure 6.11, that can be used to decide if a specification type can be used in oblivious conditionals. We say θ is *mergeable* if $\curlywedgeuparrow\theta \rhd \widehat{\texttt{ite}}$, with witness $\widehat{\texttt{ite}}$ of type $\widehat{\mathbb{B}}{\rightarrow}\theta{\rightarrow}\theta{\rightarrow}\theta$. We will write $\curlywedgeuparrow\theta$ when we do not care about the witness. This witness can be thought of as a generalized, drop-in replacement of `mux`: we simply translate `mux` to the derived $\widehat{\texttt{ite}}$ if the result type is mergeable. The case of Ψ-type captures this intuition: we first join the public views, and reshape all branches to this common public view, before we select the correct one privately using `mux`. This rule looks up the necessary methods from the context of join structures $\mathcal{S}_\sqcup$. Other cases are straightforward: we simply fall back to `mux` for primitive types, and the derivation for product and function types are done congruently.

6.3.3 Introduction and Elimination Structures

An ADT is manipulated by its introduction and elimination forms. To successfully lift a public program using ADTs, we need structures to explain how the primitive operations

of its ADTs are handled in their OADT counterparts. Thus, an OADT $\widehat{\mathtt{T}}$ can *optionally* be equipped with an *introduction-structure* (intro-structure) and an *elimination-structure* (elim-structure), defined below. These structures are optional because some programs only consume ADTs, without constructing any new ADT values (and vice versa): a function that checks membership in a list only requires an elim-structure on lists, for example. Intuitively, the axioms of these structures require the introduction and elimination methods of an OADT to behave like those of the corresponding ADT. This is formalized using a pair of logical refinement relations on values ($\mathcal{V}_n[\![\cdot]\!]$) and expressions ($\mathcal{E}_n[\![\cdot]\!]$); these relations are formally defined in Section 6.3.6.

Definition 6.3.3 (intro-structure)**.** An intro-structure of an OADT $\widehat{\mathtt{T}}$ for ADT $\mathtt{T}$, with global definition `data T = C η`, consists of a set of functions $\widehat{\mathtt{C}}_i$, each corresponding to a constructor $\mathtt{C}_i$. The type of $\widehat{\mathtt{C}}_i$ is $\theta_i \to \Psi\widehat{\mathtt{T}}$, where $\lfloor \theta_i \rfloor = \eta_i$ (note that DT-ADT guarantees that η_i is a simple type). The particular θ_i an intro-structure uses is determined by the author of that structure.

Each $\widehat{\mathtt{C}}_i$ is required to logically refine the corresponding constructor (A-I_1): given any values $\mathtt{v} : \lfloor \theta \rfloor$ and $\mathtt{v}' : \theta$, if $(\mathtt{v}, \mathtt{v}') \in \mathcal{V}_n[\![\theta]\!]$, then $(\mathtt{C}_i\ \ \mathtt{v}, \widehat{\mathtt{C}}_i\ \ \mathtt{v}') \in \mathcal{E}_n[\![\Psi\widehat{\mathtt{T}}]\!]$.

Definition 6.3.4 (elim-structure)**.** An elim-structure of an OADT $\widehat{\mathtt{T}}$ for ADT $\mathtt{T}$, with global definition `data T = C η`, consists of a family of functions $\widehat{\mathtt{match}}_\alpha$, indexed by the possible return types. The type of $\widehat{\mathtt{match}}_\alpha$ is $\Psi\widehat{\mathtt{T}} \to \overline{(\theta \to \alpha)} \to \alpha$, where $\lfloor \theta_i \rfloor = \eta_i$ for each θ_i in the function arguments corresponding to alternatives.

Each $\widehat{\mathtt{match}}_\alpha$ is required to logically refine the pattern matching expression, specialized with ADT $\mathtt{T}$ and return type α. The sole axiom of this structure (A-E_1) only considers return type α being a specification type: given values $\mathtt{v}_i : \eta_i$, $\langle \mathtt{k}, \widehat{\mathtt{v}} \rangle : \widehat{\mathtt{T}}\ \mathtt{k}$, $\lambda \mathtt{x} \Rightarrow \mathtt{e}_i : \lfloor \theta_i \rfloor \to \lfloor \alpha \rfloor$ and $\lambda \mathtt{x} \Rightarrow \mathtt{e}'_i : \theta_i \to \alpha$, if $\mathtt{r}\ \ \mathtt{k}\ \ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{C}_i\ \ \mathtt{v}_i$ and $(\lambda \mathtt{x} \Rightarrow \mathtt{e}_i, \lambda \mathtt{x} \Rightarrow \mathtt{e}'_i) \in \mathcal{V}_n[\![\theta_i \to \alpha]\!]$ then $([\mathtt{v}_i/\mathtt{x}]\,\mathtt{e}_i, \widehat{\mathtt{match}}\ \ \langle \mathtt{k}, \widehat{\mathtt{v}} \rangle\ \ \overline{(\lambda \mathtt{x} \Rightarrow \mathtt{e}')}) \in \mathcal{E}_n[\![\alpha]\!]$.

The types of the oblivious introduction and elimination forms in these structures are only required to be compatible with the public counterparts. The programmers can choose which specific OADTs to use according to their desired privacy policy. Figure 6.4 shows the constructors and pattern matching functions for $\widehat{\mathtt{list}}_{\leq}$.

$$\boxed{\theta \rightarrowtail \theta' \rhd \uparrow}$$

$$\frac{}{\theta \rightarrowtail \theta \rhd \lambda \mathtt{x} \Rightarrow \mathtt{x}} \qquad \frac{}{\mathbb{B} \rightarrowtail \widehat{\mathbb{B}} \rhd \lambda \mathtt{x} \Rightarrow \widehat{\mathbb{B}} \#\mathtt{s}\ \mathtt{x}} \qquad \frac{(\widehat{\mathtt{T}}, \mathtt{T}, \mathtt{s}, \mathtt{r}, \nu, \preccurlyeq) \in \mathcal{S}_\omega}{\mathtt{T} \rightarrowtail \Psi\widehat{\mathtt{T}} \rhd \lambda \mathtt{x} \Rightarrow \langle \nu\ \mathtt{x}, \mathtt{s}\ (\nu\ \mathtt{x})\ \mathtt{x} \rangle}$$

$$\frac{\uparrow : \Psi\widehat{\mathtt{T}} \to \Psi\widehat{\mathtt{T}}' \in \mathcal{S}_\uparrow}{\Psi\widehat{\mathtt{T}} \rightarrowtail \Psi\widehat{\mathtt{T}}' \rhd \uparrow} \qquad \frac{\theta_1 \rightarrowtail \theta_1' \rhd \uparrow_1 \qquad \theta_2 \rightarrowtail \theta_2' \rhd \uparrow_2}{\theta_1 \times \theta_2 \rightarrowtail \theta_1' \times \theta_2' \rhd \lambda \mathtt{x} \Rightarrow (\uparrow_1 (\pi_1\ \mathtt{x}), \uparrow_2 (\pi_2\ \mathtt{x}))}$$

$$\frac{\theta_1' \rightarrowtail \theta_1 \rhd \uparrow_1 \qquad \theta_2 \rightarrowtail \theta_2' \rhd \uparrow_2}{\theta_1 \to \theta_2 \rightarrowtail \theta_1' \to \theta_2' \rhd \lambda \mathtt{x} \Rightarrow \lambda \mathtt{y} \Rightarrow \uparrow_2 (\mathtt{x}\ (\uparrow_1 \mathtt{y}))}$$

Figure 6.12. Coercion

The elim-structure of an OADT consists of a family of destructors, whose return type α does not necessarily range over all types. For example, $\widehat{\mathtt{match}}_\alpha$ of $\widehat{\mathtt{list}}_\leq$, $\widehat{\mathtt{list}}_\leq \#\mathtt{match}$ in Figure 6.4, requires α to be a mergeable type, due to the use of $\widehat{\mathtt{match}}$, which imposes a restriction similarly to `mux`. Such constraints on α are automatically inferred and enforced.

6.3.4 Coercion Structures

As discussed in Section 6.1, we may need to convert an oblivious type to another, either due to a mismatch from input to output, or due to its lack of certain structures. For example, $\widehat{\mathtt{list}}_=$ does not have join structure, so if the branches of an oblivious conditional has type $\Psi\widehat{\mathtt{list}}_=$, they should be coerced to $\Psi\widehat{\mathtt{list}}_\leq$, when such a coercion is available.

Two compatible OADTs may form a *coercion-structure*, shown below.

Definition 6.3.5 (coercion-structure)**.** A coercion-structure of a pair of compatible OADTs $\widehat{\mathtt{T}}$ and $\widehat{\mathtt{T}}'$ for $\mathtt{T}$, with public view type τ and τ' respectively, consists of a coercion function $\uparrow$ of type $\Psi\widehat{\mathtt{T}} \to \Psi\widehat{\mathtt{T}}'$.

The coercion should produce an equivalent value (A-C_1): given values $\mathtt{v} : \mathtt{T}$, $\langle \mathtt{k}, \widehat{\mathtt{v}} \rangle : \Psi\widehat{\mathtt{T}}$ and $\langle \mathtt{k}', \widehat{\mathtt{v}}' \rangle : \Psi\widehat{\mathtt{T}}'$, if $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{v}$ and $\uparrow\langle \mathtt{k}, \widehat{\mathtt{v}} \rangle \longrightarrow^* \langle \mathtt{k}', \widehat{\mathtt{v}}' \rangle$, then $\mathtt{r}\ \mathtt{k}'\ \widehat{\mathtt{v}}' \longrightarrow^* \mathtt{v}$.

This structure only defines the coercion between two Ψ-types. Figure 6.12 generalizes the coercion relation to any (compatible) specification types. We say θ is *coercible* to θ' if $\theta \rightarrowtail \theta' \rhd \uparrow$, with witness $\uparrow$ of type $\theta \to \theta'$. We may write $\theta \rightarrowtail \theta'$ when we do not care about

the witness. The rules of this relation are straightforward. The context of coercion structures $\mathcal{S}_\uparrow$ and the context of OADT structures $\mathcal{S}_\omega$ are used to look up the necessary methods in the corresponding rules. The rule for coercing a function type is contravariant. Note that we can always coerce a public type to an OADT by running the section function, and the public view can be selected by the view function in the OADT structure.

6.3.5 Declarative Lifting

With these Ψ-structures, we define a declarative lifting relation, which describes what the lifting procedure is allowed to derive at a high level. This lifting relation is given by the judgment $\mathcal{S}; \mathcal{L}; \Sigma; \Gamma \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}$. It is read as the expression $\mathtt{e}$ of type $\lfloor\theta\rfloor$ is lifted to the expression $\dot{\mathtt{e}}$ of target type θ, under various contexts. The Ψ-structure context $\mathcal{S}$ consists of the set of OADT-structures ($\mathcal{S}_\omega$), join-structures ($\mathcal{S}_\sqcup$), intro-structures ($\mathcal{S}_I$), elim-structures ($\mathcal{S}_E$) and coercion-structures ($\mathcal{S}_\uparrow$), respectively. The global definition context Σ is the same as the one used in the typing relation. The local context Γ is also similar to the one in the typing relation, but it keeps track of the target types of local variables instead of source types. Finally, the lifting context $\mathcal{L}$ consists of entries of the form $\mathtt{x} : \theta \rhd \dot{\mathtt{x}}$, which associates the global function $\mathtt{x}$ of type $\lfloor\theta\rfloor$ with a generated function $\dot{\mathtt{x}}$ of the target type θ. A single global function may have multiple target types, i.e., multiple private versions, either specified by the users or by the callsites. For example, $\mathcal{L}$ may contain $\mathtt{filter} : \Psi\widehat{\mathtt{list}}_{\leq} \to \hat{\mathbb{Z}} \to \Psi\widehat{\mathtt{list}}_{\leq} \rhd \widehat{\mathtt{filter}}_1$ and $\mathtt{filter} : \Psi\widehat{\mathtt{list}}_{=} \to \hat{\mathbb{Z}} \to \Psi\widehat{\mathtt{list}}_{\leq} \rhd \widehat{\mathtt{filter}}_2$.

Figure 6.13 shows a selection of rules of the declarative lifting relation (the full rules are in Appendix B.1). We elide most contexts as they are fixed, and simply write $\Gamma \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}$ for brevity. Most rules are simply congruences and similar to typing rules. L-Fun outsources the lifting of a function call to the lifting context. L-If$_2$ handles the case when the condition is lifted to an oblivious boolean by delegating the translation to the mergeability relation. Similarly, L-Ctor$_2$ and L-Match$_2$ query the contexts of the intro-structures and elim-structures, and use the corresponding instances as the drop-in replacement, when we are constructing or destructing Ψ-types. Lastly, L-Coerce coerces an expression nondeterministically using the coercion relation.

$\boxed{\Gamma \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}}$

$$\textsc{L-Lit}\ \frac{}{\Gamma \vdash \mathtt{b} : \mathbb{B} \rhd \mathtt{b}} \qquad \textsc{L-Var}\ \frac{\mathtt{x} : \theta \in \Gamma}{\Gamma \vdash \mathtt{x} : \theta \rhd \mathtt{x}} \qquad \textsc{L-Fun}\ \frac{\mathtt{x} : \theta \rhd \dot{\mathtt{x}} \in \mathcal{L}}{\Gamma \vdash \mathtt{x} : \theta \rhd \dot{\mathtt{x}}} \qquad \textsc{L-Abs}\ \frac{\mathtt{x} : \theta_1, \Gamma \vdash \mathtt{e} : \theta_2 \rhd \dot{\mathtt{e}}}{\Gamma \vdash \lambda \mathtt{x} : \lfloor \theta_1 \rfloor \Rightarrow \mathtt{e} : \theta_1 \to \theta_2 \rhd \lambda \mathtt{x} : \theta_1 \Rightarrow \dot{\mathtt{e}}}$$

$$\textsc{L-App}\ \frac{\Gamma \vdash \mathtt{e}_2 : \theta_1 \to \theta_2 \rhd \dot{\mathtt{e}}_2 \qquad \Gamma \vdash \mathtt{e}_1 : \theta_1 \rhd \dot{\mathtt{e}}_1}{\Gamma \vdash \mathtt{e}_2\ \ \mathtt{e}_1 : \theta_2 \rhd \dot{\mathtt{e}}_2\ \ \dot{\mathtt{e}}_1}$$

$$\textsc{L-Let}\ \frac{\Gamma \vdash \mathtt{e}_1 : \theta_1 \rhd \dot{\mathtt{e}}_1 \qquad \mathtt{x} : \theta_1, \Gamma \vdash \mathtt{e}_2 : \theta_2 \rhd \dot{\mathtt{e}}_2}{\Gamma \vdash \mathtt{let\ x\ =\ e_1\ in\ e_2} : \theta_2 \rhd \mathtt{let\ x\ =\ \dot{e}_1\ in\ \dot{e}_2}}$$

$$\textsc{L-If}_1\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathbb{B} \rhd \dot{\mathtt{e}}_0 \\ \Gamma \vdash \mathtt{e}_1 : \theta \rhd \dot{\mathtt{e}}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \theta \rhd \dot{\mathtt{e}}_2\end{array}}{\Gamma \vdash \mathtt{if\ e_0\ then\ e_1\ else\ e_2} : \theta \rhd \mathtt{if\ \dot{e}_0\ then\ \dot{e}_1\ else\ \dot{e}_2}}$$

$$\textsc{L-If}_2\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \widehat{\mathbb{B}} \rhd \dot{\mathtt{e}}_0 \qquad \curlywedge\theta \rhd \widehat{\mathtt{ite}} \\ \Gamma \vdash \mathtt{e}_1 : \theta \rhd \dot{\mathtt{e}}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \theta \rhd \dot{\mathtt{e}}_2\end{array}}{\Gamma \vdash \mathtt{if\ e_0\ then\ e_1\ else\ e_2} : \theta \rhd \widehat{\mathtt{ite}}\ \ \dot{\mathtt{e}}_0\ \ \dot{\mathtt{e}}_1\ \ \dot{\mathtt{e}}_2} \qquad \textsc{L-Ctor}_1\ \frac{\mathtt{data\ T\ =\ \overline{C\ \eta}} \in \Sigma \qquad \Gamma \vdash \mathtt{e} : \eta_i \rhd \dot{\mathtt{e}}}{\Gamma \vdash \mathtt{C}_i\ \ \mathtt{e} : \mathtt{T} \rhd \mathtt{C}_i\ \ \dot{\mathtt{e}}}$$

$$\textsc{L-Ctor}_2\ \frac{\widehat{\mathtt{C}}_i : \theta_i \to \Psi\widehat{\mathtt{T}} \in \mathcal{S}_I \qquad \Gamma \vdash \mathtt{e} : \theta_i \rhd \dot{\mathtt{e}}}{\Gamma \vdash \mathtt{C}_i\ \ \mathtt{e} : \Psi\widehat{\mathtt{T}} \rhd \widehat{\mathtt{C}}_i\ \ \dot{\mathtt{e}}} \qquad \textsc{L-Match}_1\ \frac{\mathtt{data\ T\ =\ \overline{C\ \eta}} \in \Sigma \qquad \Gamma \vdash \mathtt{e}_0 : \mathtt{T} \rhd \dot{\mathtt{e}}_0 \qquad \forall i.\ \mathtt{x} : \eta_i, \Gamma \vdash \mathtt{e}_i : \theta' \rhd \dot{\mathtt{e}}_i}{\Gamma \vdash \mathtt{match\ e_0\ with\ \overline{C\ x \Rightarrow e}} : \theta' \rhd \mathtt{match\ \dot{e}_0\ with\ \overline{C\ x \Rightarrow \dot{e}}}}$$

$$\textsc{L-Match}_2\ \frac{\widehat{\mathtt{match}} : \Psi\widehat{\mathtt{T}} \to \overline{(\theta \to \theta')} \to \theta' \in \mathcal{S}_E \qquad \Gamma \vdash \mathtt{e}_0 : \Psi\widehat{\mathtt{T}} \rhd \dot{\mathtt{e}}_0 \qquad \forall i.\ \mathtt{x} : \theta_i, \Gamma \vdash \mathtt{e}_i : \theta' \rhd \dot{\mathtt{e}}_i}{\Gamma \vdash \mathtt{match\ e_0\ with\ \overline{C\ x \Rightarrow e}} : \theta' \rhd \widehat{\mathtt{match}}\ \ \dot{\mathtt{e}}_0\ \ \overline{(\lambda \mathtt{x} : \theta \Rightarrow \dot{\mathtt{e}})}}$$

$$\textsc{L-Coerce}\ \frac{\Gamma \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}} \qquad \theta \rightarrowtail \theta' \rhd \uparrow}{\Gamma \vdash \mathtt{e} : \theta' \rhd \uparrow\dot{\mathtt{e}}}$$

Figure 6.13. Selected declarative lifting rules

This lifting relation in Figure 6.13 only considers one expression. In practice, the users specify a set of functions and their target types to lift. The result of our lifting procedure is a lifting context $\mathcal{L}$ which maps these functions and target types to the corresponding generated functions, as well as any other functions and the inferred target types that these functions depend on. The global context Σ is also extended with the definitions of the generated functions. To make this more clear, we say a lifting context is *derivable*, denoted by $\vdash \mathcal{L}$, if and only if, for any $\mathtt{x} : \theta \triangleright \dot{\mathtt{x}} \in \mathcal{L}$, $\mathtt{fn\ x{:}}\lfloor\theta\rfloor\ \mathtt{=}\ \mathtt{e} \in \Sigma$ and $\mathtt{fn}\ \dot{\mathtt{x}}\mathtt{{:}}\theta\ \mathtt{=}\ \dot{\mathtt{e}} \in \Sigma$ for some $\mathtt{e}$ and $\dot{\mathtt{e}}$, such that $\mathcal{S}; \mathcal{L}; \Sigma; \cdot \vdash \mathtt{e} : \theta \triangleright \dot{\mathtt{e}}$. In other words, any definitions of the lifted functions in $\mathcal{L}$ can be derived from the lifting relation in Figure 6.13. Note that the derivation of a function definition is under a lifting context with possibly an entry of this function itself. This is similar to the role of global context in type checking, as Taypsi supports mutually recursive functions. The goal of our algorithm (Section 6.4) is then to find such a derivable lifting context that includes the user-specified liftings.

6.3.6 Logical Refinement

The correctness of the lifting procedure is framed as a *logical refinement* between expressions of specification types and those of simple types; this relationship is defined as a step-indexed logical relation [61]. As is common, this relation is defined via a pair of set-valued type denotations: a value interpretation $\mathcal{V}_n[\![\theta]\!]$ and an expression interpretation $\mathcal{E}_n[\![\theta]\!]$. We say an expression $\mathtt{e}'$ of type θ refines $\mathtt{e}$ of type $\lfloor\theta\rfloor$ (within n steps) if $(\mathtt{e}, \mathtt{e}') \in \mathcal{E}_n[\![\theta]\!]$. In other words, $\mathtt{e}'$ preserves the behavior of $\mathtt{e}$, in that if $\mathtt{e}'$ terminates at a value, $\mathtt{e}$ must terminate at an equivalent value. The equivalence between values is dictated by $\mathcal{V}_n[\![\theta]\!]$.

Figure 6.14 shows the complete definition of the logical relation. All pairs in the relations must be closed and well-typed, i.e., their interpretations have the forms:

$$\mathcal{V}_n[\![\theta]\!] = \{\ (\mathtt{v}, \mathtt{v}') \mid \cdot \vdash \mathtt{v} : \lfloor\theta\rfloor \wedge \cdot \vdash \mathtt{v}' : \theta \wedge \dots\ \}$$
$$\mathcal{E}_n[\![\theta]\!] = \{\ (\mathtt{e}, \mathtt{e}') \mid \cdot \vdash \mathtt{e} : \lfloor\theta\rfloor \wedge \cdot \vdash \mathtt{e}' : \theta \wedge \dots\ \}$$

For brevity, we leave this requirement implicit in Figure 6.14.

$\boxed{\mathcal{V}_n[\![\theta]\!]}$

$$\mathcal{V}_n[\![\mathbb{1}]\!] = \mathcal{V}_n[\![\mathbb{B}]\!] = \mathcal{V}_n[\![\mathtt{T}]\!] = \{\ (\mathtt{v}, \mathtt{v}')\ |\ 0 < n \implies \mathtt{v} = \mathtt{v}'\ \}$$

$$\mathcal{V}_n[\![\widehat{\mathbb{B}}]\!] = \{\ (\mathtt{b}, \mathtt{[b']})\ |\ 0 < n \implies \mathtt{b} = \mathtt{b}'\ \}$$

$$\mathcal{V}_n[\![\Psi\widehat{\mathtt{T}}]\!] = \{\ (\mathtt{v}, \langle \mathtt{k}, \hat{\mathtt{v}} \rangle)\ |\ 0 < n \implies \mathtt{r\ k\ \hat{v}} \longrightarrow^* \mathtt{v}\ \}$$

$$\mathcal{V}_n[\![\theta_1 \times \theta_2]\!] = \{\ ((\mathtt{v}_1, \mathtt{v}_2), (\mathtt{v}_1', \mathtt{v}_2'))\ |\ (\mathtt{v}_1, \mathtt{v}_1') \in \mathcal{V}_n[\![\theta_1]\!] \wedge (\mathtt{v}_2, \mathtt{v}_2') \in \mathcal{V}_n[\![\theta_2]\!]\ \}$$

$$\mathcal{V}_n[\![\theta_1 \to \theta_2]\!] = \left\{ (\lambda \mathtt{x}{:}\lfloor\theta_1\rfloor \Rightarrow \mathtt{e}, \lambda \mathtt{x}{:}\theta_1 \Rightarrow \mathtt{e}') \;\middle|\; \begin{array}{r} \forall i < n.\ \forall (\mathtt{v}, \mathtt{v}') \in \mathcal{V}_i[\![\theta_1]\!]. \\ (\mathtt{[v/x]e}, \mathtt{[v'/x]e}') \in \mathcal{E}_i[\![\theta_2]\!] \end{array} \right\}$$

$\boxed{\mathcal{E}_n[\![\theta]\!]}$

$$\mathcal{E}_n[\![\theta]\!] = \left\{ (\mathtt{e}, \mathtt{e}') \;\middle|\; \forall i < n.\ \forall \mathtt{v}'.\ \mathtt{e}' \longrightarrow^i \mathtt{v}' \implies \exists \mathtt{v}.\ \mathtt{e} \longrightarrow^* \mathtt{v} \wedge (\mathtt{v}, \mathtt{v}') \in \mathcal{V}_{n-i}[\![\theta]\!] \right\}$$

Figure 6.14. A logical relation for refinement

The definitions are mostly standard. The most interesting case is the value interpretation of Ψ-type: we say the pair of a public view and an oblivious value of an OADT is equivalent to a public value of the corresponding ADT when the oblivious value can be retracted to the public value. Intuitively, an encrypted value is equivalent to the value it decrypts to. The base cases of the value interpretation are also guarded by the condition that we still have steps left, i.e., greater than 0. This requirement maintains the pleasant property that the interpretations $\mathcal{V}_0[\![\theta]\!]$ and $\mathcal{E}_0[\![\theta]\!]$ are total relations on closed values and expressions, respectively, of type θ. The proof also uses a straightforward interpretation of typing context, $\mathcal{G}_n[\![\Gamma]\!]$, whose definition is in Appendix B.1.

This relation also gives rise to a semantic characterization of the lifting context. We say a lifting context is n-valid, denoted by $\vDash_n \mathcal{L}$, if and only if, for any $\mathtt{x} : \theta \rhd \dot{\mathtt{x}} \in \mathcal{L}$, $(\mathtt{x}, \dot{\mathtt{x}}) \in \mathcal{E}_n[\![\theta]\!]$. If $\vDash_n \mathcal{L}$ for any n, we say $\mathcal{L}$ is valid, denoted by $\vDash \mathcal{L}$. The validity is essentially a semantic correctness of $\mathcal{L}$.

6.3.7 Metatheory of Lifting

The first key property of the lifting relation is well-typedness, which guarantees the security of translated programs, thanks to Theorem 6.2.1.

Theorem 6.3.1 (Regularity of declarative lifting)**.** *Suppose* $\mathcal{L}$ *is well-typed and* $\mathcal{S}; \mathcal{L}; \Sigma; \Gamma \vdash \mathsf{e} : \theta \triangleright \dot{\mathsf{e}}$*. We have* $\Sigma; \lfloor \Gamma \rfloor \vdash \mathsf{e} : \lfloor \theta \rfloor$ *and* $\Sigma; \Gamma \vdash \dot{\mathsf{e}} : \theta$*.*

Our lifting relation ensures that lifted expressions refine source expressions in fewer than n steps, as long as every lifted program in $\mathcal{L}$ is also semantically correct in fewer than n steps. As is common in logical relation proofs, this proof requires a more general theorem about open terms.

Theorem 6.3.2 (Correctness of declarative lifting of closed terms)**.** *Suppose* $\mathcal{S}; \mathcal{L}; \Sigma; \cdot \vdash \mathsf{e} : \theta \triangleright \dot{\mathsf{e}}$ *and* $\vDash_n \mathcal{L}$*. We have* $(\mathsf{e}, \dot{\mathsf{e}}) \in \mathcal{E}_n[\![\theta]\!]$*.*

Finally, Theorem 6.3.3 provides a strong result of the correctness of our translation. Any lifting context that is derived using the rules of Figure 6.13 is semantically correct. In other words, if every pair of source program and lifted program in $\mathcal{L}$ are in our lifting relation, they also satisfy our refinement criteria.

Theorem 6.3.3 (Correctness of declarative lifting)**.** $\vdash \mathcal{L}$ *implies* $\vDash \mathcal{L}$*.*

Our notion of logical refinement only provides partial correctness guarantees, as can be seen in the definition of $\mathcal{E}_n[\![\cdot]\!]$. As a result, the lifting relation does not guarantee equi-termination: it is possible that a lifted program will diverge when the source program terminates. This can occur when an `if` is replaced by a `mux`: since the latter fully executes both branches, this effectively changes the semantics of a conditional from a lazy evaluation strategy to an eager strategy. Using a public value to bound the recursion depth in order to guarantee termination is a common practice in data-oblivious computation, for the reasons discussed in Section 6.1. While the public view of an OADT naturally serves as a measure in many cases, including all of the case studies and benchmarks in our evaluation, in theory it is possible for a user to provide a policy to a function that results in a nonterminating lifted version. In

this situation, users must either specify a different policy, or rewrite the functions to recurse on a different argument, e.g., a fuel value.

All the proofs in this section are available in Appendix B.3.

6.4 Algorithmic Lifting

Figure 6.15 presents the overall workflow of our lifting algorithm. This algorithm starts with a set of *goals*, i.e., pairs of source functions tagged with the `%lift` keywords and their desired specification types. We then run our lifting algorithm on all the functions in these goals, as well as any functions they depend on, transforming each of these functions to an oblivious version parameterized by *typed macros* and type variables, along with a set of constraints over these type variables. After solving the constraints, we obtain a set of type assignments for each function. Note that a single function may have multiple type assignments, one for each occurrence in a goal and callsite. For example, `filter` may have the type assignment for the goal $\Psi\widehat{\texttt{list}}_{\leq} \to \hat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}}_{\leq}$ generated by `%lift`, and the assignment for $\Psi\widehat{\texttt{list}}_{\leq} \to \mathbb{Z} \to \Psi\widehat{\texttt{list}}_{\leq}$ generated by the call in `filter5` from Section 6.1. Finally, we generate the private versions of all the lifted functions by instantiating their type variables and expanding away any macros. The lifting context from the last section is simply these lifted functions and their generated private versions.

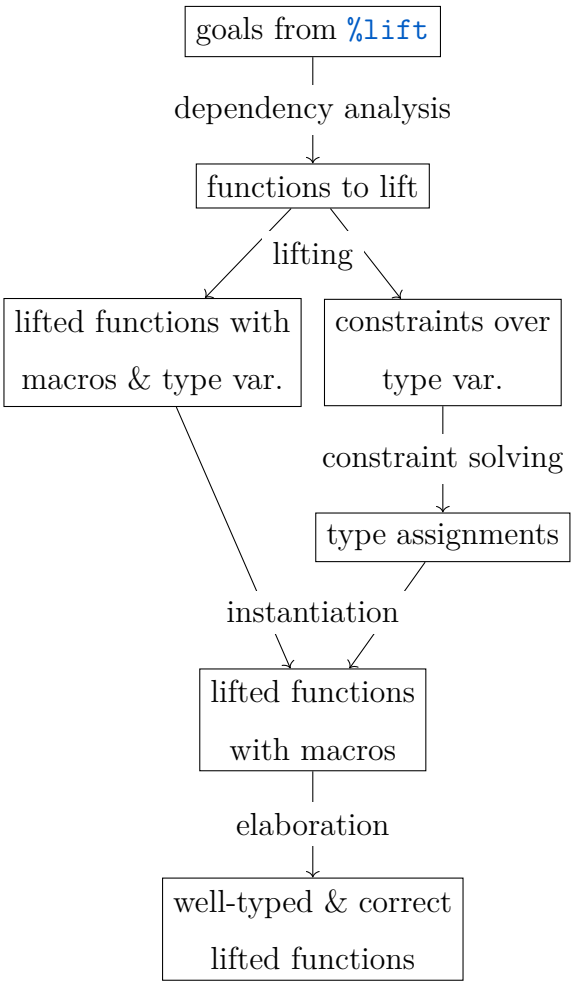


Figure 6.15. Translation pipeline

The lifting algorithm is defined using the judgment $\Sigma; \Gamma \vdash \mathtt{e} : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}} \mid \mathcal{C}$. It reads as the source expression $\mathtt{e}$ of type η is lifted to the target expression $\dot{\mathtt{e}}$ whose type is a *type variable* $\mathtt{X}$ as a placeholder for the specification type, and generates constraints $\mathcal{C}$. The source expression $\mathtt{e}$ is required to be in *administrative normal form* (ANF) [51], which is guaranteed by our type checker. In particular, type annotations are added to `let`-bindings, and the body of every `let` is either another `let` or a variable. Importantly, this means the last expression of a sequence of `let` must be a variable. The output of this algorithm is an expression $\dot{\mathtt{e}}$ containing macros (which will be discussed shortly), and the constraints $\mathcal{C}$. Unlike the declarative rules, this algorithm keeps track of the source type η, which is used to restrict the range of the type variables. Consequently, every entry of the typing context Γ has the form $\mathtt{x} : \eta \sim \mathtt{X}$, meaning that local variable $\mathtt{x}$ has type η in the source program and type $\mathtt{X}$ in the target program. For example, after the lifting algorithm has processed the function arguments of `filter` in Figure 6.1, the typing context contains entries $\mathtt{xs} : \mathtt{list} \sim \mathtt{X}_1$ and $\mathtt{y} : \mathbb{Z} \sim \mathtt{X}_2$, with freshly generated type variables $\mathtt{X}_1$ and $\mathtt{X}_2$.

The typed macros, defined in Figure 6.16, are an essential part of the output of the lifting algorithm, and permit a form of ad-hoc polymorphism, that allows the algorithm to cleanly separate constraint solving from program generation. These macros take types as parameters and elaborate to expressions, under the contexts $\mathcal{S}$, $\mathcal{L}$ and Σ implicitly. These macros are effectively thin "wrappers" of their corresponding language constructs and the previously defined relations. The conditional macro `%ite`, for example, corresponds to the `if` expression, but the condition may be oblivious. The constructor macro `%C` is a "smart" constructor that may construct a Ψ-type. The pattern matching macro `%match` is similar to `%C` but for eliminating a type compatible with an ADT. Lastly, `%↑` and `%x` is simply a direct wrapper of the mergeable relation and the lifting context $\mathcal{L}$, respectively. Note that the derivation of these macro are completely determined by the type parameters.

Figure 6.17 defines the constraints used in the algorithm, where θ^+ is the specification types extended with type variables. The constraint $\mathtt{X} \in [\eta]$ means type variable $\mathtt{X}$ belongs to the compatibility class of η. In other words, $\lfloor \mathtt{X} \rfloor = \eta$. Each macro is accompanied by a constraint on its type parameters. These constraints mean that the corresponding macros are resolvable. More formally, this means they can elaborate to some expressions according to the

$$\boxed{\%\texttt{ite}(\theta_0,\theta;e_0,e_1,e_2) \triangleright \dot{e}}$$

$$\frac{}{\%\texttt{ite}(\mathbb{B},\theta;e_0,e_1,e_2) \triangleright \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2} \qquad \frac{\curlywedge\theta \triangleright \widehat{\texttt{ite}}}{\%\texttt{ite}(\widehat{\mathbb{B}},\theta;e_0,e_1,e_2) \triangleright \widehat{\texttt{ite}}\ e_0\ e_1\ e_2}$$

$$\boxed{\%C(\theta,\theta';e) \triangleright \dot{e}}$$

$$\frac{\texttt{data}\ T = \overline{C\ \eta} \in \Sigma}{\%C_i(\eta_i,T;e) \triangleright C_i\ e} \qquad \frac{\widehat{C}_i : \theta_i \to \Psi\widehat{T} \in \mathcal{S}_I}{\%C_i(\theta_i,\Psi\widehat{T};e) \triangleright \widehat{C}_i\ e}$$

$$\boxed{\%\texttt{match}(\theta_0,\overline{\theta},\theta';e_0,\overline{e}) \triangleright \dot{e}}$$

$$\frac{\texttt{data}\ T = \overline{C\ \eta} \in \Sigma}{\%\texttt{match}(T,\overline{\eta},\theta';e_0,\overline{e}) \triangleright \texttt{match}\ e_0\ \texttt{with}\ \overline{C\ x \Rightarrow e}}$$

$$\frac{\widehat{\texttt{match}} : \Psi\widehat{T} \to \overline{(\theta \to \theta')} \to \theta' \in \mathcal{S}_E}{\%\texttt{match}(\Psi\widehat{T},\overline{\theta},\theta';e_0,\overline{e}) \triangleright \widehat{\texttt{match}}\ e_0\ \overline{(\lambda x{:}\theta \Rightarrow e)}}$$

$$\boxed{\%\uparrow(\theta,\theta';e) \triangleright \dot{e}} \qquad \boxed{\%x(\theta) \triangleright \dot{e}}$$

$$\frac{\theta \rightarrowtail \theta' \triangleright \uparrow}{\%\uparrow(\theta,\theta';e) \triangleright \uparrow e} \qquad \frac{x : \theta \triangleright \dot{x} \in \mathcal{L}}{\%x(\theta) \triangleright \dot{x}}$$

Figure 6.16. Typed macros

CONSTRAINTS

$$c \quad ::= \quad X \in [\eta] \mid \theta^+ = \theta^+$$
$$\mid \quad \%\texttt{ite}(\theta^+,\theta^+) \mid \%C(\theta^+,\theta^+) \mid \%\texttt{match}(\theta^+,\overline{\theta^+},\theta^+) \mid \%\uparrow(\theta^+,\theta^+) \mid \%x(\theta^+)$$

Figure 6.17. Constraints

rules in Figure 6.16 for any expression arguments. As a result, after solving all constraints and concretizing the type variables, all macros in the lifted expression $\dot{e}$ can be fully elaborated away.

Figure 6.18 shows a selection of lifting algorithm rules (the full rules are in Appendix B.2). Coercions only happen when we lift variables, as in A-VAR. This works because the source program is in ANF, so each expression is bound to a variable which has the opportunity to get coerced. For example, the argument to a function or constructor, in A-APP and A-CTOR, is always a variable in ANF, and recursively lifting it allows the application of

$\boxed{\Gamma \vdash \mathtt{e} : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}} \mid \mathcal{C}}$

A-Lit
$$\frac{}{\Gamma \vdash \mathtt{b} : \mathbb{B} \sim \mathtt{X} \rhd \mathtt{b} \mid \mathtt{X} = \mathbb{B}}$$

A-Var
$$\frac{\mathtt{x} : \eta \sim \mathtt{X} \in \Gamma}{\Gamma \vdash \mathtt{x} : \eta \sim \mathtt{X}' \rhd \%\uparrow(\mathtt{X},\mathtt{X}';\mathtt{x}) \mid \%\uparrow(\mathtt{X},\mathtt{X}')}$$

A-Fun
$$\frac{\mathtt{fn}\ \mathtt{x}{:}\eta\ =\ \mathtt{e} \in \Sigma}{\Gamma \vdash \mathtt{x} : \eta \sim \mathtt{X} \rhd \%\mathtt{x}(\mathtt{X}) \mid \%\mathtt{x}(\mathtt{X})}$$

A-Abs
$$\frac{\mathtt{X}_1, \mathtt{X}_2 \text{ fresh} \qquad \mathtt{x} : \eta_1 \sim \mathtt{X}_1, \Gamma \vdash \mathtt{e} : \eta_2 \sim \mathtt{X}_2 \rhd \dot{\mathtt{e}} \mid \mathcal{C}}{\Gamma \vdash \lambda \mathtt{x}{:}\eta_1 \Rightarrow \mathtt{e} : \eta_1 \to \eta_2 \sim \mathtt{X} \rhd \lambda \mathtt{x}{:}\mathtt{X}_1 \Rightarrow \dot{\mathtt{e}} \mid \mathtt{X}_1 \in [\eta_1], \mathtt{X}_2 \in [\eta_2], \mathtt{X} = \mathtt{X}_1 \to \mathtt{X}_2, \mathcal{C}}$$

A-App
$$\frac{\mathtt{X}_1 \text{ fresh} \qquad \mathtt{x}_2 : \eta_1 \to \eta_2 \sim \mathtt{X} \in \Gamma \qquad \Gamma \vdash \mathtt{x}_1 : \eta_1 \sim \mathtt{X}_1 \rhd \dot{\mathtt{e}}_1 \mid \mathcal{C}}{\Gamma \vdash \mathtt{x}_2\ \ \mathtt{x}_1 : \eta_2 \sim \mathtt{X}_2 \rhd \mathtt{x}_2\ \ \dot{\mathtt{e}}_1 \mid \mathtt{X}_1 \in [\eta_1], \mathtt{X} = \mathtt{X}_1 \to \mathtt{X}_2, \mathcal{C}}$$

A-Let
$$\frac{\mathtt{X}_1 \text{ fresh} \qquad \Gamma \vdash \mathtt{e}_1 : \eta_1 \sim \mathtt{X}_1 \rhd \dot{\mathtt{e}}_1 \mid \mathcal{C}_1 \qquad \mathtt{x} : \eta_1 \sim \mathtt{X}_1, \Gamma \vdash \mathtt{e}_2 : \eta_2 \sim \mathtt{X}_2 \rhd \dot{\mathtt{e}}_2 \mid \mathcal{C}_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{:}\eta_1\ =\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : \eta_2 \sim \mathtt{X}_2 \rhd \mathtt{let}\ \mathtt{x}{:}\mathtt{X}_1\ =\ \dot{\mathtt{e}}_1\ \mathtt{in}\ \dot{\mathtt{e}}_2 \mid \mathtt{X}_1 \in [\eta_1], \mathcal{C}_1, \mathcal{C}_2}$$

A-If
$$\frac{\mathtt{x}_0 : \mathbb{B} \sim \mathtt{X}_0 \in \Gamma \qquad \Gamma \vdash \mathtt{e}_1 : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}}_1 \mid \mathcal{C}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}}_2 \mid \mathcal{C}_2}{\Gamma \vdash \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 : \eta \sim \mathtt{X} \rhd \%\mathtt{ite}(\mathtt{X}_0,\mathtt{X};\mathtt{x}_0,\dot{\mathtt{e}}_1,\dot{\mathtt{e}}_2) \mid \%\mathtt{ite}(\mathtt{X}_0,\mathtt{X}),\mathcal{C}_1,\mathcal{C}_2}$$

A-Ctor
$$\frac{\mathtt{data}\ \mathtt{T}\ =\ \overline{\mathtt{C}\ \eta} \in \Sigma \qquad \mathtt{X}_i \text{ fresh} \qquad \Gamma \vdash \mathtt{x} : \eta_i \sim \mathtt{X}_i \rhd \dot{\mathtt{e}} \mid \mathcal{C}}{\Gamma \vdash \mathtt{C}_i\ \ \mathtt{x} : \mathtt{T} \sim \mathtt{X} \rhd \%\mathtt{C}_i(\mathtt{X}_i,\mathtt{X};\dot{\mathtt{e}}) \mid \mathtt{X}_i \in [\eta_i], \%\mathtt{C}_i(\mathtt{X}_i,\mathtt{X}), \mathcal{C}}$$

A-Match
$$\frac{\begin{array}{c}\mathtt{data}\ \mathtt{T}\ =\ \overline{\mathtt{C}\ \eta} \in \Sigma \\ \overline{\mathtt{X}} \text{ fresh} \qquad \mathtt{x}_0 : \mathtt{T} \sim \mathtt{X}_0 \in \Gamma \qquad \forall i.\ \mathtt{x} : \eta_i \sim \mathtt{X}_i, \Gamma \vdash \mathtt{e}_i : \eta' \sim \mathtt{X}' \rhd \dot{\mathtt{e}}_i \mid \mathcal{C}_i\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{match}\ \mathtt{x}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{e}} : \eta' \sim \mathtt{X}' \rhd \%\mathtt{match}(\mathtt{X}_0,\overline{\mathtt{X}},\mathtt{X}';\mathtt{x}_0,\overline{\dot{\mathtt{e}}}) \\ \quad \mid \overline{\mathtt{X} \in [\eta]}, \%\mathtt{match}(\mathtt{X}_0,\overline{\mathtt{X}},\mathtt{X}'), \overline{\mathcal{C}}\end{array}}$$

Figure 6.18. Selected algorithmic lifting rules

A-Var. On the other hand, the top-level program is always in `let`-binding form, whose last expression is always a variable too, allowing coercion of the whole program. However, not all variables are subject to coercions: the function $\mathtt{x}_2$ in A-App, the condition $\mathtt{x}_0$ in A-If and the discriminee $\mathtt{x}_0$ in A-Match are kept as they are, for example. Coercing these variables would be unnecessary and undesirable. For example, coercing the condition in a conditional only makes the generated program more expensive: there is no reason to coerce from $\mathbb{B}$ to $\widehat{\mathbb{B}}$, and use `mux` instead of `if`. Another key invariant we enforce in our algorithmic rules is that every fresh variable is "guarded" by a compatibility class constraint. For example, in A-Abs, the freshly generated variables $\mathtt{X}_1$ and $\mathtt{X}_2$ belong to the classes η_1 and η_2, respectively. This constraint ensures that every type variable can be finitely enumerated, as every compatibility class is a finite set, bounded by the number of available OADTs. As a result, constraint solving in our context is decidable. Finally, if an expression is translated to a macro, a corresponding constraint is added to ensure this macro is resolvable.

We use the judgment $\mathcal{S}; \mathcal{L}; \Sigma; \sigma \vDash \mathcal{C}$ to mean the assignment σ satisfies a set of constraints $\mathcal{C}$, under the context of Ψ-structure, lifting context and global definition context. The constraints generated by our lifting algorithm use type variables `X` as placeholders for the target type of the function being lifted. To solve a goal with a particular target type θ, we add a constraint to $\mathcal{C}$ that equates the placeholder with the stipulated type, i.e., `X = θ`. Our constraint solver then attempts to find type assignments that satisfy the constraints in $\mathcal{C}$; the resulting assignment is used to generate private versions of all the functions in the set of goals, as well as the accompanying lifting context.

6.4.1 Constraint Solving

At a high level, our solver reduces all constraints, except for function call constraints (`%x`), to quantifier-free formulas in a finite domain theory, which can be efficiently solved using an off-the-shelf solver. Function call constraints are recursively solved once their type arguments have been concretized by discharging the other constraints. When a function call constraint is unsatisfiable, we add a new refutation constraint and invoke the solver again to find a new instantiation of type parameters. As an example of this process, in

order to ascribe `filter` the type $\Psi\widehat{\texttt{list}_{=}} \to \hat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}_{\leq}}$, we first add the constraint $\texttt{X} = \Psi\widehat{\texttt{list}_{=}} \to \hat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}_{\leq}}$ to the constraints generated by the lifting algorithm $\cdot \vdash \ldots : \texttt{list} \to \mathbb{Z} \to \texttt{list} \sim \texttt{X} \rhd \dot{\texttt{e}} \mid \mathcal{C}$. Solving the other constraints may concretize the type variable of function call constraint `%filter(X)`, i.e., the type of the recursive call to `filter`, to $\texttt{\%filter}(\Psi\widehat{\texttt{list}_{=}} \to \hat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}_{\leq}})$. Recursively solving this subgoal assuming the original goal is solved, i.e., extending the lifting context with the original goal, results in immediate success, as the subgoal is simply in the lifting context. On the other hand, if the type of the recursive call is instantiated as $\texttt{\%filter}(\Psi\widehat{\texttt{list}_{=}} \to \hat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}_{=}})$, the same constraints generated by lifting `filter` are solved, with an additional constraint $\texttt{X} = \Psi\widehat{\texttt{list}_{=}} \to \hat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}_{=}}$. However, this set of constraints is unsatisfiable, as $\widehat{\texttt{list}_{=}}$ has no join structure, so we add a refutation constraint to the context that forces the solver to not generate this assignment again. In general, the type of the recursive call to `filter` may be concretized to any types compatible with $\texttt{list} \to \mathbb{Z} \to \texttt{list}$. The number of such compatible types is bounded, as the number of arguments of this function and the number of OADTs are themselves bounded. The function `filter` has $3 \times 2 \times 3 = 18$ possible type assignments. In the worst case scenario, the algorithm eventually terminates after exhausting all 18 combinations.

More formally, the constraint solving algorithm uses two maps. $\mathcal{F}$ maps function names to the generated constraints with a type variable `X` as the placeholder for the potential specification type, resulting from the lifting algorithm: every entry of $\mathcal{F}$ has the form $\texttt{f} \mapsto (\texttt{X}, \mathcal{C}, \mathcal{C}')$. The generated constraints are partitioned into $\mathcal{C}$, which consists of all constraints except for function call constraints, and $\mathcal{C}'$, which consists of the functional call constraints (`%x`). Initially, $\mathcal{F}$ consists of all functions collected from the keyword `%lift` and the functions they depend on. $\mathcal{M}$ maps a pair of function name `f` and its target type θ, called a *goal*, to a type assignment σ: each entry has the form $(\texttt{f}, \theta) \mapsto \sigma$. Initially $\mathcal{M}$ is empty.

The constraint solving algorithm takes an initial $\mathcal{F}$ and $\mathcal{M}$, and a goal, and returns an updated $\mathcal{M}$ that consists of the type assignments for the goal and all the subgoals this goal depends on. The constraint solver is applied to all goals from `%lift`, and the final $\mathcal{M}$ is the union of all returned $\mathcal{M}$. With $\mathcal{M}$, we can generate functions from the type assignments for each goal. The global context is subsequently extended by these generated functions, and

Inputs: Constraint map $\mathcal{F}$, type assignment map $\mathcal{M}$, function name $\mathtt{f}$ and target type θ
Output: Updated type assignment map $\mathcal{M}'$
function SOLVE($\mathcal{F}$, $\mathcal{M}$, $\mathtt{f}$, θ)
  if $(\mathtt{f}, \theta) \mapsto \sigma \in \mathcal{M}$ **then return** $\mathcal{M}$
  $(\mathtt{X}, \mathcal{C}, \mathcal{C}') \leftarrow \mathcal{F}(\mathtt{f})$
  $\phi \leftarrow$ LOWER($\mathtt{X} = \theta, \mathcal{C}$)
  if QF-FD-SOLVE(ϕ) returns unsat **then fail**
  else if QF-FD-SOLVE(ϕ) returns sat with σ **then**
    if there is a $\%\mathtt{g}(\theta') \in \mathcal{C}'$ s.t. SOLVE($\mathcal{F}$, $\mathcal{M}[(\mathtt{f}, \theta) \mapsto \sigma]$, $\mathtt{g}$, $\sigma(\theta')$) fails **then**
      SOLVE($\mathcal{F}[\mathtt{f} \mapsto (\mathtt{X}, \mathcal{C} \cup \{\, \theta' \neq \sigma(\theta') \,\}, \mathcal{C}')]$, $\mathcal{M}$, $\mathtt{f}$, θ)
    else
      $\{\, (\mathtt{f}, \theta) \mapsto \sigma \,\} \cup \bigcup$ SOLVE($\mathcal{F}$, $\mathcal{M}[(\mathtt{f}, \theta) \mapsto \sigma]$, $\mathtt{g}$, $\sigma(\theta')$) for all $\%\mathtt{g}(\theta') \in \mathcal{C}'$

Figure 6.19. Constraint solving algorithm

the final lifting context is constructed by pairing each goal with its corresponding generated function.

Figure 6.19 presents a naive algorithm for constraint solving. The subroutine LOWER is used to reduce all constraints except for function call constraints to formulas in the *quantifier-free finite domain theory* (`QF_FD`), then an off-the-shelf solver (Z3 [40]) is used to solve them. LOWER first decomposes all compatibility class constraints, i.e., $\mathtt{X} \in [\eta]$, into *atomic* classes. For example, $\mathtt{X} \in [\mathtt{list} \to \mathbb{Z}]$ is decomposed into $\mathtt{X}_1 \in [\mathtt{list}]$ and $\mathtt{X}_2 \in [\mathbb{Z}]$, with $\mathtt{X}$ substituted by $\mathtt{X}_1 \to \mathtt{X}_2$ in all other constraints. Then the newly generated compatibility class constraints can be reduced to a disjunction of all possible specification types in this class. For example, $\mathtt{X}_1 \in [\mathtt{list}]$ reduces to $\mathtt{X}_1 = \mathtt{list} \vee \mathtt{X}_1 = \Psi\widehat{\mathtt{list}}_{\leq} \vee \mathtt{X}_1 = \Psi\widehat{\mathtt{list}}_{=}$, if $\widehat{\mathtt{list}}_{\leq}$ and $\widehat{\mathtt{list}}_{=}$ are the only OADTs for `list`. After this step, all type variables are compatible with an atomic simple type, so we can also decompose other constraints into a set of base cases according to the rules of their relations. Note that while mergeability is not one of the constraints generated by the lifting algorithm, such constraints still arise from `%ite` and `%match`. When multiple assignments are valid for a particular type variable, we prefer the "cheaper", i.e., more permissive, solution. For example, we prefer public type `list` over OADTs if possible, and prefer $\widehat{\mathtt{list}}_{=}$ over $\widehat{\mathtt{list}}_{\leq}$. We encode these preferences as *soft constraints*, and assign a bigger penalty to more restrictive types. The penalty is inferred by

analyzing the coercion relations: a more permissive type can be coerced to a more restrictive type but not the other way around, because the more restrictive type hides more information.

Once other constraints are solved, the type parameters of function call constraints are concretized. The algorithm is then recursively applied to this subgoal, i.e., the functional call paired with the concretized type, assuming the original goal is solved by extending $\mathcal{M}$ with the original goal and its type assignments. This handles potential (mutual) recursion. If a subgoal fails, its target type will be added as a refutation to the corresponding constraints, and backtrack.

It is easy to see that this algorithm terminates: every recursive call to SOLVE either adds a refutation that reduces the search space of the non-function-call constraints, or extends $\mathcal{M}$ which is finitely bounded by the number of functions and the number of OADTs, hence reducing the search space of function-call constraints.

Our implementation maintains a more sophisticated state of $\mathcal{M}$ and applies several optimizations to reduce the number of calls to the external solver. We also exploit the incremental solver of Z3 to help with performance.

6.4.2 Metatheory of Algorithmic Lifting

The lifting algorithm enjoys a soundness theorem with respect to the declarative lifting relation. As a result, our algorithm inherits the well-typedness and correctness properties of the declarative version. The statement of this theorem follows how the algorithm is used: if the generated constraints, equating the function type variable with the specification type, are satisfiable by the type assignment σ, instantiating the lifted expression with σ and elaborating the macros results in a target expression that is valid under the declarative lifting relation:

Theorem 6.4.1 (Soundness of algorithmic lifting)**.** *Suppose* $\Sigma; \cdot \vdash \mathsf{e} : \eta \sim \mathtt{X} \rhd \dot{\mathsf{e}} \mid \mathcal{C}$*. Given a specification type* θ*, if* $\mathcal{S}; \mathcal{L}; \Sigma; \sigma \vDash \mathtt{X} = \theta, \mathcal{C}$*, then* $\sigma(\dot{\mathsf{e}})$ *elaborates to an expression* $\dot{\mathsf{e}}'$*, such that* $\mathcal{S}; \mathcal{L}; \Sigma; \cdot \vdash \mathsf{e} : \theta \rhd \dot{\mathsf{e}}'$*.*

The proof of this theorem is available in Appendix B.3.

6.5 Implementation

Our compilation pipeline takes as input a source program, including any OADTs, Ψ-structures, and macros (e.g., `%lift`), in the public fragment of Taypsi and privacy policies (i.e., security-type signatures) for all target functions. After typing the source program using a bidirectional type checker, our lifting pass generates secure versions of the specified functions and their dependencies, using Z3 [40] as its constraint solver. The resulting Taypsi functions are translated into Oil (Section 5.3), an ML-style functional language equipped with oblivious arrays and secure array operations: OADTs are converted to serialized versions which are stored in secure arrays, and all oblivious operations are translated into secure array operations. After applying some optimizations, our pipeline outputs an OCaml library providing secure implementations of all the specified functions, including section and retraction functions for encrypting private data and decrypting the results of a joint computation. After linking this library to a *driver* that provides the necessary cryptographic primitives (i.e., secure integer arithmetic), programmers can build secure MPC applications on top of this API. The evaluation in Section 6.6 uses a driver implemented using the popular open-source EMP toolkit [52].

6.5.1 Optimizations

Our implementation of Taypsi implements three optimizations which further improve the performance of the programs it generates.

The *smart array* optimization supports zero-cost array slicing and concatenation, and eliminates redundant operations over the serialized representation of oblivious data. To reduce the overhead of constructing and destructing oblivious data, our implementation does not create new arrays when performing array slicing and concatenation. It allows the results to follow the original structure in Taypsi for as long as possible, until a `mux` forces them to be flattened. Conceptually, the smart arrays delay these operations, performing them all at once when flattening is required. On the other hand, to eliminate the redundant cryptographic operations, one observation underlying this optimization is that evaluating a `mux` whose branches are encrypted versions of publicly-known values is

unnecessary: `mux [b] (`$\widehat{\mathbb{B}}$`#s true) (`$\widehat{\mathbb{B}}$`#s false)` is equivalent to `[b]`, for example. This situation frequently occurs in map-like functions, where the constructor used in each branch of a function is publicly known. Under the hood, the serialized encoding of the result of `map` uses a boolean tag to indicate which constructor was used to build it, i.e., `Nil` or `Cons`; this boolean is determined by the tag of the input list, e.g., `mux [tag] [true] [false]`. Of course, the tag used in each branch is publicly known: `map` always returns `Nil` if the input list is empty, and returns a `Cons` otherwise. Thus, we can safely reuse the `[tag]` of the input list to label the result of `map`, for similar reasons as the previous example. The smart array optimization exploits this observation by marking when section functions are applied to public values instead of, for example, immediately evaluating $\widehat{\mathbb{B}}$`#s true` to the encrypted value `[true]`. Then, when performing a `mux`, the smart array first checks if both branches are "fake" private values, safely reducing the `mux` to its private condition if so, without actually performing any cryptographic operations. In addition, we keep track of whether a value is "arbitrary", e.g., used for padding. We can simply return a branch of a `mux` if the other branch is an arbitrary value.

The *reshape guard* optimization instruments reshape instances to first check if the public views of two private values are identical, omitting the reshape operation if so. Reshaping OADTs to the same public view is a common scenario, especially when the partial order defined on a public view type is a total order. For example, the join of the public views of $\widehat{\texttt{list}}_{\leq}$ is the maximum one, which is always one of these public views. Therefore, reshaping the private lists of this maximum length should not require any additional work.

The *memoization* optimization caches the sizes of the private representation of data in order to avoid recalculating this information, which is needed to create and slice oblivious arrays. Similar to Taype, invoking these size computations in a recursive function can potentially introduce asymptotic slowdown. Taype eliminates these repeated computations using a tupling optimization that merges the size functions with the section and retraction functions. However, in Taypsi, this size calculation can happen in any arbitrary private functions, in addition to section and retraction functions, making tupling optimization brittle to apply. Instead, Taypsi memoizes the map from public views to the sizes of the private representation to avoid recalculation. If an OADT uses integer public view, then the

memoization technique is standard using a hash table for retrieving the calculated sizes. On the other hand, it is not efficient to use ADT public views as keys of a hash table. We instead automatically embed the sizes within a public view type itself, and rewrite its introduction and elimination forms and all Taypsi programs that use them accordingly. For example, a Peano number public view `peano` is augmented as follows.

```
data peano = peano_memo × ℕ
data peano_memo = Zero | Succ peano
```

Whenever we need to calculate the size of a private representation using this public view `peano`, we simply project out its size (of type ℕ), without recalculating the size function.

6.6 Evaluation

Our evaluation considers the following research questions:

RQ1 How does the performance of Taypsi's transformation-based approach compare to the dynamic enforcement strategy of Taype?

RQ2 What is the compilation overhead of Taypsi's translation strategy?

6.6.1 Microbenchmark Performance

To answer **RQ1**, we have evaluated the performance of a set of microbenchmarks compiled with both Taypsi and Taype. Both approaches are equipped with optimizations that are unique to their enforcement strategies: Taypsi's reshape guard optimization is not applicable to Taype, and Taype features an *early tape* optimization that does not make sense for Taypsi.[11] Our evaluation also includes a version of Taype that implements Taypsi's smart array optimization (Taype-SA), in order to provide a comparison of the two approaches at their full potential.

Our benchmarks are a superset of the benchmarks from Section 5.5. Figure 6.20 presents the experimental results.[12] These experiments fix the public views of private lists and trees

[11]↑Taype also implements a tupling optimization, but this is analogous to Taypsi's memoization optimization.
[12]↑All results are averaged across 5 runs, on an M1 MacBook Pro with 16 GB memory. All parties run on the same host with local network communication.

Benchmark	Taype (ms)	Taype-SA (ms)	Taypsi (ms)	
`elem_1000`†	8.15	8.11	8.02	(98.47%, 98.89%)
`hamming_1000`†	15.09	15.21	14.46	(95.79%, 95.04%)
`euclidean_1000`†	67.43	67.55	67.32	(99.84%, 99.66%)
`dot_prod_1000`†	66.12	66.19	66.41	(100.43%, 100.33%)
`nth_1000`†	11.98	12.05	12.04	(100.54%, 99.93%)
`map_1000`	2139.55	5.07	5.14	(0.24%, 101.44%)
`filter_200`	**failed**	**failed**	86.86	(N/A, N/A)
`insert_200`	5796.69	88.92	88.07	(1.52%, 99.04%)
`insert_list_100`	**failed**	**failed**	4667.66	(N/A, N/A)
`append_100`	4274.7	45.09	44.18	(1.03%, 97.99%)
`take_200`	169.07	3.05	3.09	(1.83%, 101.15%)
`flat_map_200`	**failed**	**failed**	7.3	(N/A, N/A)
`span_200`	13529.34	124.79	91.22	(0.67%, 73.09%)
`partition_200`	**failed**	**failed**	176.49	(N/A, N/A)
`elem_16`†	446.81	459.1	404.9	(90.62%, 88.19%)
`prob_16`†	13082.52	12761.7	12735.16	(97.34%, 99.79%)
`map_16`	4414.69	262.14	215.67	(4.89%, 82.27%)
`filter_16`	8644.14	452.04	433.7	(5.02%, 95.94%)
`swap_16`	**failed**	**failed**	4251.36	(N/A, N/A)
`path_16`	**failed**	6657.07	894.88	(N/A, 13.44%)
`insert_16`	83135.81	8093.81	1438.87	(1.73%, 17.78%)
`bind_8`	21885.65	494.98	532.86	(2.43%, 107.65%)
`collect_8`	**failed**	**failed**	143.38	(N/A, N/A)

Figure 6.20. Running times for each benchmark in milliseconds. The Taypsi column also reports the percentage of running time relative to Taype and Taype-SA. A **failed** entry indicates the benchmark either timed out after 5 minutes or exceeded the memory bound of 8 GB. List and tree benchmarks appear above and below the double line, respectively.

to be their maximum length and maximum depth, respectively; the suffix of each benchmark name indicates the public view used. The benchmarks annotated with † simply traverse the data type in order to produce a primitive value, e.g., an integer; these include membership (`elem`), hamming distance (`hamming`), minimum euclidean distance (`euclidean`), dot product (`dot_prod`), secure index look up (`nth`) and computing the probability of an event given a probability tree diagram (`prob`). The programs generated by Taype, Taype-SA and Taypsi all exhibit similar performance on these benchmarks. The remaining benchmarks all construct structured data values, i.e., the sort of application on which Taypsi is expected to shine.

In addition to standard list operations, the list benchmarks include insertion into a sorted list (`insert`) and insertion of a list of elements into a sorted list (`insert_list`) (both lists have public view 100). The tree examples include a filter function that removes all nodes (including any subtrees) greater than a given private integer (`filter`), swapping subtrees if the node matches a private integer (`swap`), computing a subtree reached following a list of "going left" and "going right" directions (`path`), insertion into a binary search tree (`insert`), replacing the leaves of a tree with a given tree (`bind`), and collecting all nodes smaller than a private integer into a list (`collect`).

Dynamic policy enforcement either fails to finish within 5 minutes or exceeds an 8 GB memory bound on almost half of the last set of benchmarks, due to the exponential blowup discussed in Section 6.1. For those benchmarks that do finish, Taypsi's enforcement strategy results in a fraction of the total execution time compared to Taype. Compared to the version of Taype using smart arrays, Taypsi still performs comparably or better, although the gap is somewhat narrowed: functions like `map` do not suffer from exponential blowup, so these benchmarks benefit mostly from the smart array optimization. In summary, these results demonstrate that a static enforcement strategy performs considerably better than a dynamic one on many benchmarks, and works roughly as well on the remainder.

6.6.2 Impact of Optimization

To evaluate the performance impact of Taypsi's three optimizations, we conducted an ablation study on their effect. The results, shown in Figure 6.21, indicate that our smart array optimization is the most important, providing up to almost 800x speedup in the best case. As suggested by Figure 6.20, this optimization also helps significantly with the performance of Taype, although not enough to outweigh the exponential blowup innate in its dynamic approach. The other optimizations also improve performance, albeit not as significantly. As our memoization pass caches public views of arbitrary type, we have also conducted an ablation study for these examples using ADT public views instead: the list examples use Peano number to encode the maximum length of a list, and the tree examples use the upper

Benchmark	No SA (ms)		No RG (ms)		No Memo (ms)	
elem_1000	18.37	(2.29x)	8.06	(1.0x)	17.76	(2.21x)
hamming_1000	51.73	(3.58x)	14.53	(1.01x)	35.5	(2.46x)
euclidean_1000	79.07	(1.17x)	67.31	(1.0x)	76.36	(1.13x)
dot_prod_1000	87.77	(1.32x)	66.15	(1.0x)	77.33	(1.16x)
nth_1000	22.69	(1.88x)	12.18	(1.01x)	20.53	(1.7x)
map_1000	2106.43	(409.89x)	139.91	(27.23x)	37.71	(7.34x)
filter_200	5757.28	(66.29x)	93.93	(1.08x)	114.7	(1.32x)
insert_200	255.43	(2.9x)	94.61	(1.07x)	89.32	(1.01x)
insert_list_100	22806.87	(4.89x)	5186.07	(1.11x)	4771.28	(1.02x)
append_100	4226.32	(95.66x)	50.79	(1.15x)	61.77	(1.4x)
take_200	169.45	(54.91x)	12.92	(4.19x)	4.68	(1.52x)
flat_map_200	5762.63	(789.08x)	16.99	(2.33x)	60.03	(8.22x)
span_200	5924.1	(64.95x)	99.83	(1.09x)	120.09	(1.32x)
partition_200	11528.0	(65.32x)	185.16	(1.05x)	231.06	(1.31x)
elem_16	433.73	(1.07x)	404.05	(1.0x)	402.15	(0.99x)
prob_16	13019.56	(1.02x)	12746.24	(1.0x)	12731.89	(1.0x)
map_16	4410.84	(20.45x)	635.18	(2.95x)	213.96	(0.99x)
filter_16	8674.71	(20.0x)	1131.02	(2.61x)	440.16	(1.01x)
swap_16	8671.52	(2.04x)	5471.4	(1.29x)	4246.39	(1.0x)
path_16	9108.54	(10.18x)	1083.21	(1.21x)	888.95	(0.99x)
insert_16	19101.36	(13.28x)	2151.83	(1.5x)	1432.92	(1.0x)
bind_8	19647.83	(36.87x)	870.93	(1.63x)	534.3	(1.0x)
collect_8	11830.6	(82.51x)	152.29	(1.06x)	186.92	(1.3x)

Figure 6.21. Impact of turning off the smart array (No SA), reshape guard (No RG), and public view memoization (No Memo) optimizations. Each column presents running time in milliseconds and the slowdown relative to that of the fully optimized version reported in Figure 6.20.

bound of the spines. In this study (Figure 6.22), we observe up to 9 times speed up in the list examples, with minimal regression in tree examples.

6.6.3 Compilation Overhead

To measure the overhead of Taypsi's use of an external solver to resolve constraints, we have profiled the compilation of a set of larger programs drawn from Taype's benchmark suite. The first two benchmark suites (List and Tree) in Figure 6.23 include all the microbenchmarks from previous section. The next benchmark, List (stress), consists of the

Benchmark	Base (ms)	No Memoization (ms)	
`elem_1000`	13.45	18.2	(1.35x)
`hamming_1000`	25.98	36.35	(1.4x)
`euclidean_1000`	73.22	77.07	(1.05x)
`dot_prod_1000`	77.65	77.92	(1.0x)
`nth_1000`	17.8	20.89	(1.17x)
`map_1000`	19.95	42.53	(2.13x)
`filter_200`	87.22	118.54	(1.36x)
`insert_200`	89.01	89.77	(1.01x)
`insert_list_100`	4719.13	4809.16	(1.02x)
`append_100`	45.05	63.83	(1.42x)
`take_200`	4.08	5.17	(1.27x)
`flat_map_200`	7.32	69.1	(9.45x)
`span_200`	92.73	124.06	(1.34x)
`partition_200`	176.68	238.27	(1.35x)
`elem_16`	425.41	416.13	(0.98x)
`prob_16`	12772.3	12762.84	(1.0x)
`map_16`	268.07	255.66	(0.95x)
`filter_16`	494.38	485.96	(0.98x)
`swap_16`	4407.25	4329.39	(0.98x)
`path_16`	940.83	944.69	(1.0x)
`insert_16`	1603.88	1770.76	(1.1x)
`bind_8`	553.0	585.06	(1.06x)
`collect_8`	143.79	187.25	(1.3x)

Figure 6.22. Impact of turning off the public view memoization (No Memoization) optimization for the examples using ADT public views. The No Memoization column also reports the slowdown relative to Base, the fully optimized version.

same microbenchmarks as List with 5 additional list OADTs. The purpose of this synthetic suite is to examine the impact of the number of OADTs on the search space. The remaining benchmarks represent larger, more realistic applications which demonstrate the expressivity and usability of Taypsi.

The last three columns of Figure 6.23 report the results of these experiments: total compilation time (Total), time spent on constraint solving (Solver) and the number of solver queries (#Queries). The group of columns in the middle of the table describes features that can impact the performance of our constraint-based approach: the number

Suite	#Functions	#Types	#Atoms	#Queries	Total (s)	Solver (s)
List	20	7	70	84	0.47	0.081
Tree	14	9	44	31	0.47	0.024
List (stress)	20	12	70	295	3.45	2.8
Dating	4	13	16	10	0.58	0.019
Medical Records	20	19	58	51	0.48	0.072
Secure Calculator	2	9	6	5	1.34	0.013
Decision Tree	2	13	6	16	0.28	0.016
K-means	16	11	68	86	1.62	0.95
Miscellaneous	11	7	42	47	0.26	0.065

Figure 6.23. Impact of constraint solving on compilation speed

of functions (#Functions) being translated, the number of atomic types (#Types), and the total number of atomic types used in function types (#Atoms). For example, the List benchmark features 7 atomic types: public and oblivious booleans, integers and lists, as well as an unsigned integer type (i.e., natural numbers). The number of atomic types in the function `filter : list` $\to \mathbb{Z} \to$ `list` is 3. In the worst case scenario, our constraint solving algorithm will explore every combination of types that are compatible with this signature, resulting in the constraints associated with `filter` being solved $2 * 2 * 2 = 8$ times. Exactly how many compatible types the constraint solving algorithm explores depends on many factors: the user-specified policies, the complexity of the functions, the calls to other functions and so on. We chose these 3 metrics as a coarse approximation of the solution space. Our results show that the solver overhead is quite minimal for most benchmarks, and in general solving time per query is low thanks to our encoding of constraints in an efficiently decidable logic.

6.7 Conclusion

Secure multiparty computation allows joint computation over private data from multiple parties, while keeping that data secure. Taype has considered how to make languages for MPC more accessible by allowing privacy requirements to be decoupled from functionality, relying on dynamic enforcement of polices. Unfortunately, the resulting overhead of this strategy made it difficult to scale applications manipulating structured data. This chapter

presents Taypsi, a policy-agnostic language for oblivious computation that transforms programs to instead *statically* enforce a user-provided privacy policy. The resulting programs are guaranteed to be both well-typed, and hence secure, and equivalent to the source program. Our experimental results show this strategy yields considerable performance improvements over prior approaches, while maintaining a clean separation between privacy and programmatic concerns.

7. RELATED WORK

Secure computation was first formally introduced by Yao [1] 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 [2, 62]. Those in the former category typically use protocols based on either Garbled Circuits or secret-sharing schemes [5, 6, 63]. In the realm of outsourced computation, solutions are typically based on fully homomorphic encryption [7, 8], but can also be supported by virtualization [9, 10] or secure processors [11]. Our implementation uses the EMP toolkit [52] for its secure backend, but is compatible with other solutions under the mild requirement that they implement primitives for secure integer operations.

Many high-level programming languages have been proposed that support some form of secure computation [4]. 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. To the best of our knowledge, none of these languages decouple security policies and program logic, as Taype and Taypsi do. Obliv-C [13] 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. PICCO [14, 64] is also a C-like language for secure computation, which supports C pointers to private data (possibly at private locations) and dynamic memory allocation [65]. ObliVM [15] 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* [16, 17] 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 languages do not consider mixed-mode computation. Symphony [20, 66] 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. λ_{obliv} [18] 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 approach and theirs share similar threat models and guarantees of obliviousness. Other secure computation toolchains include FairplayMP [67], Sharemind [68], CBMC-GC [69], SCVM [70], TinyGarble [71], and Frigate [72].

Several prior works have considered how to compile secure programs into more efficient secure versions. Viaduct [19, 73] is a compiler that transforms high-level programs into secure distributed versions by intelligently selecting an efficient combination of protocols for subcomputations. The HyCC toolchain [74] similarly transforms a C program into a version that combines different MPC protocols to optimize performance. TASTY [75], ABY [76], EzPC [77] and MOTION [78] are similar frameworks for enabling *mixed-protocol computation*. The HACCLE toolchain [79] uses staging to generate efficient garbled circuits from a high-level language. Compiler techniques, e.g., vectorization, have been studied for optimizing fully homomorphic encryption (FHE) applications [59, 80–83].

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 [9]. Barthe *et al.* [84] extended the formally verified CompCert compiler [85] to ensure constant time execution. Our obliviousness theorem provides a formal guarantee of a constant-time

property. Jasmin [86, 87] is a framework for implementing high-performance cryptography. It achieves constant-time security by embedding Jasmin programs into Dafny [88] which enables automated proofs of this property (and memory safety). FaCT [89] 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 into a constant-time FaCT program. In Taype and Taypsi, the programmers can simply encode programs in the conventional fragment and then convert them to oblivious programs (that are constant-time) by composing privacy policies and the standard programs.

Another popular cryptographic technique for hiding private information of data structures is oblivious RAM [26, 90, 91] (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 [92]. 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. On the other hand, certain data structures may yield asymptotically better performance if encoded using ORAM. Integrating ORAM into our systems for performance gains while maintaining strong security guarantees is a promising future direction.

Our approach of type-based information flow control to enforce obliviousness, a form of *noninterference*, follows a body of work in *security-type systems* [27, 29]. 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. While most security-type systems tag types with labels classifying the sensitivity of data, our dependent type system tags kinds instead to keep track of whether a term is oblivious. On the other hand, our leakage labels are similar to these security labels, and used to track if a term is leaky. Our notion of retraction bears some resemblance to delimited information release [93]. In a system with delimited information release, the programmers may choose to reveal some private information, similar to retraction functions in $\lambda_{\texttt{OADT}\blacksquare}$. However, our semantics guarantees retraction never releases any private information. Another difference from 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 obliviousness guarantee is a strengthened variant of *memory trace obliviousness* (MTO) [28], which itself provides stronger guarantees than most information flow type systems. Under MTO, the patterns of memory access generated by a program are required to be indistinguishable, in addition to its result. This work also proposed a language based on *Oblivious RAM* [26, 90, 91] and transformation techniques to ensure this property. However, this threat model is weaker than that of this dissertation. On the one hand, it does not consider timing channel: while memory access traces include instruction fetches, which ensures the branches of a secure conditional always run the same number of instructions, the instructions themselves can still exhibit different timing behaviors. For example, the program `if s > 0 then s := p + p else s := p * p` is secure in their model, as both branches produce the same memory access pattern (including instruction fetches), but the second branch is slower, assuming multiplication is slower than addition in the CPU. On the other hand, under MTO, adversaries cannot observe the instructions executed by the CPU. This is not the case in the MPC setting (especially in the secret-sharing-based schemes), as every party is a potential adversary that can observe instructions: `if s > 0 then s := p + 1 else s := p + 2` is accepted in their model, but an adversary in our model is able to discern if the program is computing `p+1` or `p+2`, even if they have the same timing behavior. In contrast, the traces we consider include every program state under a small-step semantics, which rules out these two examples.

Jeeves [94] and our work have a shared goal of decoupling security policies from program logic. While they both employ a similar high-level strategy of relying on the language to automatically enforce policies, their different settings result in very different solutions. In Jeeves' programming model, each piece of data is equipped with a pair of high- and low-level views: a username, for example, may have a high confidentiality view of "Alice", but a low view of "Anonymous". The language then uses the view stipulated by the privacy policy and

current execution context, ensuring that information is only visible to observers with the proper authority. In the MPC setting, however, no party is allowed to observe the private data of other parties. Thus, no party can view all the data necessary for the computation, making it impossible to compute a correct result by simply replacing data with some predetermined value, like "Anonymous".

Our oblivious types can also be viewed as a kind of refinement types [36–38]: the oblivious tree in Chapter 3 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 and to enable simpler policy specifications. Dependent type systems with large elimination can be found in many theorem provers, such as Coq [33] and Agda [35]. 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 language with nontermination is Zombie [41–43], though our goals are drastically different.

Nanevski *et al.* [95] 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 all of our oblivious calculi protect against a larger class of leaks. The 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 [96–99], 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 [100, 101].

8. CONCLUSIONS AND FUTURE WORK

Writing secure applications that do not leak private information presents challenges in several dimensions. The limited support for rich data structures in existing oblivious languages discourages users from implementing general applications. The inability to express complex policies makes it difficult to comply with real-world privacy requirements. The intermixing of application logic and privacy policies forces programmers to write specialized versions of the same program for each set of privacy requirements. This dissertation introduced various programming language techniques that tackle these problems: private structured data and complex policies can be encoded using oblivious algebraic data types, and a form of modularity that decouples privacy and programmatic concerns can be achieved using tape semantics or static program transformations. This thesis has formally developed these techniques via a family of core calculi, which it used to establish key metatheoretic results, including a strong security guarantee called obliviousness and the standard type safety property. These techniques have been implemented in the languages and compilation pipelines, Taype and Taypsi, providing an end-to-end programming environment for developing secure applications and evaluating performance and usability.

The work presented in this dissertation is a step towards wider adoption of privacy-preserving techniques, but there is still much room to make these policy-agnostic languages more practical. First, to enable a wider range of applications, we should increase the expressivity of these languages: adding support for mutable data, probabilistic computation and reactive programs that allow for interaction between participating entities in a policy-agnostic way are all important directions. Another limitation of the current approach is that OADTs can only encode tree-like structures, similar to ADTs. Supporting graph data structures will enable more secure applications. Second, while the separation of policies and application logic reduces user burden, many potential improvements to usability remain. As one example, Taype and Taypsi do not support type polymorphism, and as a result, users need to define, e.g., distinct types for lists of integers and lists of booleans. Writing OADTs and their associated methods requires programmers to be proficient in dependent types. At the same time, many of these definitions are straightforward but tedious boilerplate.

A better approach would be to derive OADTs from a higher-level predicate that specifies the publicly shared information and synthesize the associated methods. The current implement of Taypsi can generate nonterminating programs if a specified policy does not provide enough public information to bound the recursion depth. Guaranteeing equi-termination for the generated secure programs would free users from manually reasoning about termination. Third, good performance is necessary for encouraging adoption of these language techniques. The abstractions developed for the modularity described in this dissertation often come with performance penalty. Users may sometimes need to manually optimize a program to avoid some efficiency pitfalls caused by the secure semantics of `mux`, for example. Thus, additional optimization techniques are needed to generate secure programs that are as efficient as hand-crafted versions. One promising direction is to rewrite program control flow in a way that exploits publicly available information to remove secure operations.

REFERENCES

[1] A. C. Yao, "Protocols for secure computations," in *23rd Annual Symposium on Foundations of Computer Science (Sfcs 1982)*, Nov. 1982, pp. 160–164. DOI: 10.1109/SFCS.1982.38.

[2] D. Evans, V. Kolesnikov, and M. Rosulek, "A Pragmatic Introduction to Secure Multi-Party Computation," *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2-3, pp. 70–246, 2018, ISSN: 2474-1558, 2474-1566. DOI: 10.1561/3300000019. [Online]. Available: http://www.nowpublishers.com/article/Details/SEC-019.

[3] P. Laud and L. Kamm, Eds., *Applications of Secure Multiparty Computation* (Cryptology and Information Security Series volume 13). Amsterdam, Netherlands: IOS Press, 2015, 253 pp., ISBN: 978-1-61499-532-6 978-1-61499-531-9.

[4] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "SoK: General Purpose Compilers for Secure Multi-Party Computation," in *2019 2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2019, pp. 479–496. DOI: 10.1109/SP.2019.00028. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2019.00028.

[5] O. Goldreich, "Towards a theory of software protection and simulation by oblivious RAMs," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '87, New York, NY, USA: Association for Computing Machinery, Jan. 1, 1987, pp. 182–194, ISBN: 978-0-89791-221-1. DOI: 10.1145/28395.28416. [Online]. Available: http://doi.org/10.1145/28395.28416.

[6] A. Beimel, "Secret-Sharing Schemes: A Survey," in *Coding and Cryptology*, Y. M. Chee *et al.*, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, pp. 11–46, ISBN: 978-3-642-20901-7. DOI: 10.1007/978-3-642-20901-7_2.

[7] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09, New York, NY, USA: Association for Computing Machinery, May 31, 2009, pp. 169–178, ISBN: 978-1-60558-506-2. DOI: 10.1145/1536414.1536440. [Online]. Available: http://doi.org/10.1145/1536414.1536440.

[8] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A Survey on Homomorphic Encryption Schemes: Theory and Implementation," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, 79:1–79:35, Jul. 25, 2018, ISSN: 0360-0300. DOI: 10.1145/3214303. [Online]. Available: http://doi.org/10.1145/3214303.

[9] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level Non-interference for Constant-time Cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, Scottsdale, Arizona, USA: Association for Computing Machinery, Nov. 3, 2014, pp. 1267–1279, ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660283. [Online]. Available: http://doi.org/10.1145/2660267.2660283.

[10] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, "System-Level Non-interference of Constant-Time Cryptography. Part I: Model," *Journal of Automated Reasoning*, vol. 63, no. 1, pp. 1–51, Jun. 1, 2019, ISSN: 1573-0670. DOI: 10.1007/s10817-017-9441-5. [Online]. Available: https://doi.org/10.1007/s10817-017-9441-5.

[11] M. E. Hoekstra. "Intel SGX for Dummies (Intel SGX Design Objectives)." (Nov. 30, 2015), [Online]. Available: https://www.intel.com/content/www/us/en/develop/blogs/protecting-application-secrets-with-intel-sgx.html.

[12] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - a secure two-party computation system," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04, USA: USENIX Association, Aug. 13, 2004, p. 20.

[13] S. Zahur and D. Evans, "Obliv-C: A Language for Extensible Data-Oblivious Computation," 1153, 2015. [Online]. Available: https://eprint.iacr.org/2015/1153.

[14] Y. Zhang, A. Steele, and M. Blanton, "PICCO: A general-purpose compiler for private distributed computation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, New York, NY, USA: Association for Computing Machinery, Nov. 4, 2013, pp. 813–826, ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516752. [Online]. Available: https://dl.acm.org/doi/10.1145/2508859.2516752.

[15] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A Programming Framework for Secure Computation," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 359–376. DOI: 10.1109/SP.2015.29.

[16] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 655–670. DOI: 10.1109/SP.2014.48.

[17] A. Rastogi, N. Swamy, and M. Hicks, "Wys*: A DSL for Verified Secure Multi-party Computations," in *Principles of Security and Trust*, F. Nielson and D. Sands, Eds., ser. Lecture Notes in Computer Science, Springer International Publishing, 2019, pp. 99–122, ISBN: 978-3-030-17138-4. [Online]. Available: https://doi.org/10.1007/978-3-030-17138-4_5.

[18] D. Darais, I. Sweet, C. Liu, and M. Hicks, "A language for probabilistically oblivious computation," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–31, POPL Jan. 2020, ISSN: 2475-1421, 2475-1421. DOI: 10.1145/3371118. arXiv: 1711.09305. [Online]. Available: http://dl.acm.org/doi/10.1145/3371118.

[19] C. Acay, R. Recto, J. Gancher, A. C. Myers, and E. Shi, "Viaduct: An extensible, optimizing compiler for secure distributed programs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021, New York, NY, USA: Association for Computing Machinery, Jun. 19, 2021, pp. 740–755, ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454074. [Online]. Available: https://doi.org/10.1145/3453483.3454074.

[20] I. Sweet, D. Darais, D. Heath, W. Harris, R. Estes, and M. Hicks, "Symphony: Expressive Secure Multiparty Computation with Coordination," *The Art, Science, and Engineering of Programming*, vol. 7, no. 3, 14:1–14:55, Feb. 15, 2023, ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2023/7/14. [Online]. Available: https://programming-journal.org/2023/7/14/.

[21] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *1982 IEEE Symposium on Security and Privacy*, Apr. 1982, pp. 11–11. DOI: 10.1109/SP.1982.10014.

[22] Q. Ye and B. Delaware, "Oblivious algebraic data types," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, 51:1–51:29, Jan. 2022. DOI: 10.1145/3498713. [Online]. Available: https://doi.org/10.1145/3498713.

[23] Q. Ye and B. Delaware, "Taype: A Policy-Agnostic Language for Oblivious Computation," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, 147:1001–147:1025, Jun. 2023. DOI: 10.1145/3591261. [Online]. Available: https://dl.acm.org/doi/10.1145/3591261.

[24] Q. Ye and B. Delaware, "Taypsi: Static Enforcement of Privacy Policies for Policy-Agnostic Oblivious Computation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, Apr. 2024. DOI: 10.1145/3649861. [Online]. Available: https://dl.acm.org/doi/10.1145/3649861.

[25] R. Canetti, "Security and Composition of Multiparty Cryptographic Protocols," *Journal of Cryptology*, vol. 13, no. 1, pp. 143–202, Jan. 1, 2000, ISSN: 1432-1378. DOI: 10.1007/s001459910006. [Online]. Available: https://doi.org/10.1007/s001459910006.

[26] O. Goldreich, S. Micali, and A. Wigderson, "How to play ANY mental game," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '87, New York, New York, USA: Association for Computing Machinery, Jan. 1, 1987, pp. 218–229, ISBN: 978-0-89791-221-1. DOI: 10.1145/28395.28420. [Online]. Available: http://doi.org/10.1145/28395.28420.

[27] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003, ISSN: 1558-0008. DOI: 10.1109/JSAC.2002.806121.

[28] C. Liu, M. Hicks, and E. Shi, "Memory Trace Oblivious Program Execution," in *2013 IEEE 26th Computer Security Foundations Symposium*, Jun. 2013, pp. 51–65. DOI: 10.1109/CSF.2013.11.

[29] S. A. Zdancewic, "Programming languages for information security," Ph.D. dissertation, Cornell University, USA, 2002, 327 pp.

[30] W. W. Tait, "Intensional interpretations of functionals of finite type I," *The Journal of Symbolic Logic*, vol. 32, no. 2, pp. 198–212, Aug. 1967, ISSN: 0022-4812, 1943-5886. DOI: 10.2307/2271658. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-symbolic-logic/article/intensional-interpretations-of-functionals-of-finite-type-i/9F30EA199783BD797DF6FA44525F114E.

[31] G. D. Plotkin, "Lambda Definability and Logical Relations," University of Edinburgh, Memorandum SAI-RM-4, 1973. [Online]. Available: https://homepages.inf.ed.ac.uk/gdp/publications/logical_relations_1973.pdf.

[32] G. Barthe and T. Coquand, "An Introduction to Dependent Type Theory," in *Applied Semantics*, G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, Eds., Berlin, Heidelberg: Springer, 2002, pp. 1–41, ISBN: 978-3-540-45699-5. DOI: 10.1007/3-540-45699-6_1.

[33] The Coq Development Team, *The Coq Proof Assistant*, Zenodo, Jun. 2023. DOI: 10.5281/zenodo.8161141. [Online]. Available: https://zenodo.org/records/8161141.

[34] L. de Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Automated Deduction – CADE 28*, A. Platzer and G. Sutcliffe, Eds., Cham: Springer International Publishing, 2021, pp. 625–635, ISBN: 978-3-030-79876-5. DOI: 10.1007/978-3-030-79876-5_37.

[35] Agda Developers, *Agda*, Mar. 2024. [Online]. Available: https://agda.readthedocs.io/.

[36] H. Xi and F. Pfenning, "Dependent types in practical programming," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99, New York, NY, USA: Association for Computing Machinery, Jan. 1999, pp. 214–227, ISBN: 978-1-58113-095-9. DOI: 10.1145/292540.292560. [Online]. Available: http://doi.org/10.1145/292540.292560.

[37] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid Types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, New York, NY, USA: ACM, 2008, pp. 159–169, ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375602. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375602.

[38] M. Kawaguchi, P. Rondon, and R. Jhala, "Type-based data structure verification," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: Association for Computing Machinery, Jun. 2009, pp. 304–315, ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542510. [Online]. Available: http://doi.org/10.1145/1542476.1542510.

[39] N. Swamy *et al.*, "Dependent types and multi-monadic effects in F*," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16, St. Petersburg, FL, USA: Association for Computing Machinery, Jan. 2016, pp. 256–270, ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837655. [Online]. Available: http://doi.org/10.1145/2837614.2837655.

[40] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2008, pp. 337–340, ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[41] V. Sjöberg *et al.*, "Irrelevance, Heterogeneous Equality, and Call-by-value Dependent Type Systems," *Electronic Proceedings in Theoretical Computer Science*, vol. 76, pp. 112–162, Feb. 2012, ISSN: 2075-2180. DOI: 10.4204/EPTCS.76.9. arXiv: 1202.2923. [Online]. Available: http://arxiv.org/abs/1202.2923.

[42] V. Sjöberg and S. Weirich, "Programming up to Congruence," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, New York, NY, USA: Association for Computing Machinery, Jan. 2015, pp. 369–382, ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676974. [Online]. Available: http://doi.org/10.1145/2676726.2676974.

[43] V. Sjöberg, "A Dependently Typed Language with Nontermination," *Publicly Accessible Penn Dissertations*, Jan. 2015. [Online]. Available: https://repository.upenn.edu/edissertations/1137.

[44] H. Barendregt, "Introduction to generalized type systems," *Journal of Functional Programming*, vol. 1, no. 2, pp. 125–154, Apr. 1991, ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796800020025. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/introduction-to-generalized-type-systems/869991BA6A99180BF96A616894C6D710.

[45] H. P. Barendregt, "Lambda Calculi with Types," in *Handbook of Logic in Computer Science*, J. Spurr, S. Abramsky, and D. M. Gabbay, Eds., Oxford University Press, Dec. 1992, ISBN: 978-0-19-853761-8. DOI: 10.1093/oso/9780198537618.003.0002. [Online]. Available: https://doi.org/10.1093/oso/9780198537618.003.0002.

[46] B. C. Pierce, Ed., *Advanced Topics in Types and Programming Languages*. Cambridge, Mass: MIT Press, 2005, ISBN: 978-0-262-16228-9.

[47] H. P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics* (Studies in Logic and the Foundations of Mathematics v. 103), Rev. ed. Amsterdam ; New York : New York, N.Y: North-Holland ; Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, 1984, ISBN: 978-0-444-86748-3 978-0-444-87508-2.

[48] Q. Ye and B. Delaware, *Oblivious Algebraic Data Types: POPL22 Artifact*, Zenodo, Oct. 2021. DOI: 10.5281/zenodo.5652106. [Online]. Available: https://zenodo.org/record/5652106.

[49] Q. Ye and B. Delaware, *Taype: A Policy-Agnostic Language for Oblivious Computation: PLDI23 Artifact*, Zenodo, Apr. 2023. DOI: 10.5281/zenodo.7806981. [Online]. Available: https://zenodo.org/record/7806981.

[50] J. Dunfield and N. Krishnaswami, "Bidirectional Typing," *ACM Computing Surveys*, vol. 54, no. 5, 98:1–98:38, May 2021, ISSN: 0360-0300. DOI: 10.1145/3450952. [Online]. Available: https://doi.org/10.1145/3450952.

[51] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 237–247, Jun. 1993, ISSN: 0362-1340. DOI: 10.1145/173262.155113. [Online]. Available: https://doi.org/10.1145/173262.155113.

[52] X. Wang, A. J. Malozemoff, and J. Katz, *EMP-toolkit: Efficient MultiParty computation toolkit*, https://github.com/emp-toolkit, 2016.

[53] R. S. Bird, "Using circular programs to eliminate multiple traversals of data," *Acta Informatica*, vol. 21, no. 3, pp. 239–250, Oct. 1984, ISSN: 1432-0525. DOI: 10.1007/BF00264249. [Online]. Available: https://doi.org/10.1007/BF00264249.

[54] W.-N. Chin, "Towards an automated tupling strategy," in *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ser. PEPM '93, New York, NY, USA: Association for Computing Machinery, Aug. 1993, pp. 119–132, ISBN: 978-0-89791-594-6. DOI: 10.1145/154630.154643. [Online]. Available: https://doi.org/10.1145/154630.154643.

[55] R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," *Journal of the ACM*, vol. 24, no. 1, pp. 44–67, Jan. 1977, ISSN: 0004-5411. DOI: 10.1145/321992.321996. [Online]. Available: https://doi.org/10.1145/321992.321996.

[56] V. F. Turchin, R. M. Nirenberg, and D. V. Turchin, "Experiments with a supercompiler," in *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, ser. LFP '82, New York, NY, USA: Association for Computing Machinery, Aug. 1982, pp. 47–55, ISBN: 978-0-89791-082-8. DOI: 10.1145/800068.802134. [Online]. Available: https://doi.org/10.1145/800068.802134.

[57] D. J. Wu, T. Feng, M. Naehrig, and K. Lauter, "Privately Evaluating Decision Trees and Random Forests," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 335–355, Oct. 1, 2016, ISSN: 2299-0984. DOI: 10.1515/popets-2016-0043. [Online]. Available: https://petsymposium.org/popets/2016/popets-2016-0043.php.

[58] Á. Kiss, M. Naderpour, J. Liu, N. Asokan, and T. Schneider, "SoK: Modular and Efficient Private Decision Tree Evaluation," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 2, pp. 187–208, Apr. 1, 2019, ISSN: 2299-0984. DOI: 10.2478/popets-2019-0026. [Online]. Available: https://petsymposium.org/popets/2019/popets-2019-0026.php.

[59] R. Malik, V. Singhal, B. Gottfried, and M. Kulkarni, "Vectorized secure evaluation of decision forests," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021, New York, NY, USA: Association for Computing Machinery, Jun. 18, 2021, pp. 1049–1063, ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454094. [Online]. Available: https://dl.acm.org/doi/10.1145/3453483.3454094.

[60] Q. Ye and B. Delaware, *Taypsi: Static Enforcement of Privacy Policies for Policy-Agnostic Oblivious Computation: OOPSLA24 Artifact*, Zenodo, Mar. 2024. DOI: 10.5281/zenodo.10701642. [Online]. Available: https://zenodo.org/records/10701642.

[61] A. Ahmed, "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types," in *Programming Languages and Systems*, P. Sestoft, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2006, pp. 69–83, ISBN: 978-3-540-33096-7. DOI: 10.1007/11693024_6.

[62] C. Hazay and Y. Lindell, *Efficient Secure Two-Party Protocols: Techniques and Constructions* (Information Security and Cryptography). Berlin ; London: Springer, 2010, 263 pp., ISBN: 978-3-642-14302-1.

[63] U. Maurer, "Secure multi-party computation made simple," *Discrete Applied Mathematics*, Coding and Cryptography, vol. 154, no. 2, pp. 370–381, Feb. 1, 2006, ISSN: 0166-218X. DOI: 10.1016/j.dam.2005.03.020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166218X05002428.

[64] A. Rathore, M. Blanton, M. Gaboardi, and L. Ziarek, *A Formal Model for Secure Multiparty Computation*, May 2023. DOI: 10.48550/arXiv.2306.00308. arXiv: 2306.00308 [cs]. [Online]. Available: http://arxiv.org/abs/2306.00308.

[65] Y. Zhang, M. Blanton, and G. Almashaqbeh, "Implementing Support for Pointers to Private Data in a General-Purpose Secure Multi-Party Compiler," *ACM Transactions on Privacy and Security*, vol. 21, no. 2, 6:1–6:34, Dec. 2017, ISSN: 2471-2566. DOI: 10.1145/3154600. [Online]. Available: https://dl.acm.org/doi/10.1145/3154600.

[66] I. Sweet, D. Darais, D. Heath, R. Estes, W. Harris, and M. Hicks, "Symphony: A concise language model for MPC," in *Informal Proceedings of the Workshop on Foundations on Computer Secuirty (FCS)*, Jun. 2021.

[67] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: A system for secure multi-party computation," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, New York, NY, USA: Association for Computing Machinery, Oct. 2008, pp. 257–266, ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455804. [Online]. Available: https://dl.acm.org/doi/10.1145/1455770.1455804.

[68] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A Framework for Fast Privacy-Preserving Computations," in *Computer Security - ESORICS 2008*, S. Jajodia and J. Lopez, Eds., Berlin, Heidelberg: Springer, 2008, pp. 192–206, ISBN: 978-3-540-88313-5. DOI: 10.1007/978-3-540-88313-5_13.

[69] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, New York, NY, USA: Association for Computing Machinery, Oct. 2012, pp. 772–783, ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382278. [Online]. Available: https://dl.acm.org/doi/10.1145/2382196.2382278.

[70] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks, "Automating Efficient RAM-Model Secure Computation," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 623–638. DOI: 10.1109/SP.2014.46. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6956591.

[71] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 411–428. DOI: 10.1109/SP.2015.32. [Online]. Available: https://ieeexplore.ieee.org/document/7163039?denied=.

[72] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, "Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, Mar. 2016, pp. 112–127. DOI: 10.1109/EuroSP.2016.20. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7467350.

[73] C. Acay, J. Gancher, R. Recto, and A. C. Myers. "Secure Synthesis of Distributed Cryptographic Applications (Technical Report)." arXiv: 2401.04131 [cs]. (Jan. 5, 2024), [Online]. Available: http://arxiv.org/abs/2401.04131, preprint.

[74] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "HyCC: Compilation of Hybrid Protocols for Practical Secure Computation," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, (Toronto, Canada), ser. CCS '18, New York, NY, USA: ACM, 2018, pp. 847–861, ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243786. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243786.

[75] W. Henecka, S. K ögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "TASTY: Tool for automating secure two-party computations," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, Chicago, Illinois, USA: Association for Computing Machinery, Oct. 4, 2010, pp. 451–462, ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866358. [Online]. Available: http://doi.org/10.1145/1866307.1866358.

[76] D. Demmler, T. Schneider, and M. Zohner, "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," in *Proceedings 2015 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2015, ISBN: 978-1-891562-38-9. DOI: 10.14722/ndss.2015.23113. [Online]. Available: https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/aby---framework-efficient-mixed-protocol-secure-two-party-computation/.

[77] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, Jun. 2019, pp. 496–511. DOI: 10.1109/EuroSP.2019.00043. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8806756.

[78] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko, "MOTION - A Framework for Mixed-Protocol Multi-Party Computation," *ACM Transactions on Privacy and Security*, vol. 25, no. 2, 8:1–8:35, Mar. 4, 2022, ISSN: 2471-2566. DOI: 10.1145/3490390. [Online]. Available: http://doi.org/10.1145/3490390.

[79] Y. Bao *et al.*, "HACCLE: Metaprogramming for secure multi-party computation," in *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2021, New York, NY, USA: Association for Computing Machinery, Oct. 17, 2021, pp. 130–143, ISBN: 978-1-4503-9112-2. DOI: 10.1145/3486609.3487205. [Online]. Available: https://doi.org/10.1145/3486609.3487205.

[80] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, New York, NY, USA: Association for Computing Machinery, Jun. 11, 2020, pp. 546–561, ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3386023. [Online]. Available: https://dl.acm.org/doi/10.1145/3385412.3386023.

[81] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: A synthesizing compiler for vectorized homomorphic encryption," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021, New York, NY, USA: Association for Computing Machinery, Jun. 18, 2021, pp. 375–389, ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454050. [Online]. Available: https://dl.acm.org/doi/10.1145/3453483.3454050.

[82] R. Malik, K. Sheth, and M. Kulkarni, "Coyote: A Compiler for Vectorizing Encrypted Arithmetic Circuits," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, New York, NY, USA: Association for Computing Machinery, Mar. 25, 2023, pp. 118–133, ISBN: 978-1-4503-9918-0. DOI: 10.1145/3582016.3582057. [Online]. Available: https://dl.acm.org/doi/10.1145/3582016.3582057.

[83] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "HECO: Fully Homomorphic Encryption Compiler," presented at the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA: USENIX Association, Aug. 2023, pp. 4715–4732, ISBN: 978-1-939133-37-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/viand.

[84] G. Barthe *et al.*, "Formal verification of a constant-time preserving C compiler," *Proceedings of the ACM on Programming Languages*, vol. 4, 7:1–7:30, POPL Dec. 20, 2019. DOI: 10.1145/3371075. [Online]. Available: http://doi.org/10.1145/3371075.

[85] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, p. 107, Jul. 1, 2009, ISSN: 00010782. DOI: 10.1145/1538788.1538814. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1538788.1538814.

[86] J. B. Almeida *et al.*, "Jasmin: High-Assurance and High-Speed Cryptography," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, New York, NY, USA: Association for Computing Machinery, Oct. 30, 2017, pp. 1807–1823, ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134078. [Online]. Available: https://doi.org/10.1145/3133956.3134078.

[87] J. B. Almeida *et al.*, "The Last Mile: High-Assurance and High-Speed Cryptographic Implementations," in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 965–982. DOI: 10.1109/SP40000.2020.00028.

[88] K. R. M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds., Berlin, Heidelberg: Springer, 2010, pp. 348–370, ISBN: 978-3-642-17511-4. DOI: 10.1007/978-3-642-17511-4_20.

[89] S. Cauligi *et al.*, "FaCT: A DSL for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, Jun. 8, 2019, pp. 174–189, ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314605. [Online]. Available: http://doi.org/10.1145/3314221.3314605.

[90] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of the ACM*, vol. 43, no. 3, pp. 431–473, May 1, 1996, ISSN: 0004-5411. DOI: 10.1145/233551.233553. [Online]. Available: http://doi.org/10.1145/233551.233553.

[91] E. Stefanov *et al.*, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, New York, NY, USA: Association for Computing Machinery, Nov. 4, 2013, pp. 299–310, ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516660. [Online]. Available: http://doi.org/10.1145/2508859.2516660.

[92] X. S. Wang *et al.*, "Oblivious Data Structures," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, Scottsdale, Arizona, USA: Association for Computing Machinery, Nov. 3, 2014, pp. 215–226, ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660314. [Online]. Available: http://doi.org/10.1145/2660267.2660314.

[93] A. Sabelfeld and A. C. Myers, "A Model for Delimited Information Release," in *Software Security - Theories and Systems*, K. Futatsugi, F. Mizoguchi, and N. Yonezaki, Eds., ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 174–191, ISBN: 978-3-540-37621-7. [Online]. Available: https://doi.org/10.1007/978-3-540-37621-7_9.

[94] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '12*, Philadelphia, PA, USA: ACM Press, 2012, p. 85, ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103669. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2103656.2103669.

[95] A. Nanevski, A. Banerjee, and D. Garg, "Dependent Type Theory for Verification of Information Flow and Access Control Policies," *ACM Transactions on Programming Languages and Systems*, vol. 35, no. 2, 6:1–6:41, Jul. 1, 2013, ISSN: 0164-0925. DOI: 10.1145/2491522.2491523. [Online]. Available: http://doi.org/10.1145/2491522.2491523.

[96] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal Certification of Code-based Cryptographic Proofs," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Savannah, GA, USA), ser. POPL '09, New York, NY, USA: ACM, 2009, pp. 90–101, ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480894. [Online]. Available: http://doi.acm.org/10.1145/1480881.1480894.

[97] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-Aided Security Proofs for the Working Cryptographer," in *Advances in Cryptology – CRYPTO 2011*, P. Rogaway, Ed., ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 71–90, ISBN: 978-3-642-22792-9.

[98] C. Abate *et al.*, "SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, Jun. 2021, pp. 1–15. DOI: 10.1109/CSF51468.2021.00048.

[99] M. Barbosa *et al.*, "SoK: Computer-Aided Cryptography," in *2021 IEEE Symposium on Security and Privacy (SP)*, May 2021, pp. 777–795. DOI: 10.1109/SP40001.2021.00008. [Online]. Available: https://ieeexplore.ieee.org/document/9519449.

[100] M. Backes, M. Maffei, and E. Mohammadi, "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)*, K. Lodaya and M. Mahajan, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 8, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 352–363, ISBN: 978-3-939897-23-1. DOI: 10.4230/LIPIcs.FSTTCS.2010.352. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2877.

[101] H. Haagh, A. Karbyshev, S. Oechsner, B. Spitters, and P.-Y. Strub, "Computer-Aided Proofs for Multiparty Computation with Active Security," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, Jul. 2018, pp. 119–131. DOI: 10.1109/CSF.2018.00016.

A. OMITTED RULES AND PROOFS FOR TAYPE

This chapter documents the omitted rules from Chapter 5: full semantics rules (Appendix A.1), typing and kinding rules (Appendix A.2), and TAYPE-to-OIL translation rules (Appendix A.3). A proof of the totality of the translation algorithm is also included (Appendix A.4).

A.1 Semantics

Figure A.2 and Figure A.3 show the small-step operational semantics relation of core TAYPE: $\Sigma \vdash \mathtt{e} \longrightarrow \mathtt{e}'$. The global context Σ is fixed in the rules, so we elide it for brevity. We also elide type annotations in conditionals and pattern matching expressions as they are not relevant in these rules. The auxiliary relation used in S-OMATCH is exactly the same as the one in $\lambda_{\texttt{OADT}}$ (Figure 3.9). S-CTX refers to evaluation contexts $\mathcal{E}$ shown in Figure A.1.

EVALUATION CONTEXTS

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \Box \mathbin{\widehat{\times}} \tau \mid \widehat{\omega} \mathbin{\widehat{\times}} \Box \mid \Box \mathbin{\widehat{+}} \tau \mid \widehat{\omega} \mathbin{\widehat{+}} \Box \\
& \mid & \texttt{let x}:_{\iota}\tau\texttt{ = }\Box\texttt{ in e} \mid \texttt{e}\ \Box \mid \Box\ \texttt{v} \mid \texttt{C}\ \Box \mid \widehat{\texttt{T}}\ \Box \\
& \mid & \texttt{if }\Box\texttt{ then e else e} \mid \texttt{mux }\Box\texttt{ e e} \mid \texttt{mux v }\Box\texttt{ e} \mid \texttt{mux v v }\Box \\
& \mid & \widehat{\texttt{if}}\ \Box\ \texttt{then e else e} \mid \widehat{\texttt{if}}\ \texttt{v then }\Box\texttt{ else e} \mid \widehat{\texttt{if}}\ \texttt{v then v else }\Box \\
& \mid & (\Box\texttt{,e}) \mid (\texttt{v,}\Box) \mid \langle\Box\texttt{,e}\rangle \mid \langle\texttt{v,}\Box\rangle \mid \widehat{\iota}_b\texttt{<}\Box\texttt{> e} \mid \widehat{\iota}_b\texttt{<}\widehat{\omega}\texttt{>}\ \Box \\
& \mid & \Box \oplus \texttt{e} \mid \texttt{v} \oplus \Box \mid \Box \mathbin{\widehat{\oplus}} \texttt{e} \mid \texttt{v} \mathbin{\widehat{\oplus}} \Box \\
& \mid & \texttt{match }\Box\texttt{ with }\overline{\texttt{C x} \Rightarrow \texttt{e}} \mid \texttt{match }\Box\texttt{ with }(\texttt{x}_1\texttt{,x}_2)\Rightarrow\texttt{e} \\
& \mid & \widehat{\texttt{match}}\ \Box\ \texttt{with x}\Rightarrow\texttt{e|x}\Rightarrow\texttt{e} \mid \widehat{\texttt{match}}\ \Box\ \texttt{with }[\texttt{x}_1\texttt{,x}_2]\Rightarrow\texttt{e} \\
& \mid & \widehat{\mathbb{B}}\texttt{\#s }\Box \mid \widehat{\mathbb{Z}}\texttt{\#s }\Box \mid \widehat{\mathbb{Z}}\texttt{\#r }\Box \mid \texttt{tape }\Box \mid \uparrow\Box
\end{array}
$$

Figure A.1. Core TAYPE evaluation context

S-ADD and S-OADD simply evaluate the integer operations according to the denotational domain. Most rules involving promotions (e.g., S-SECINTPROM, S-APPPROM and S-ADDPROM) require the resulting expressions also get promoted, because the $\top$ leakage label should be preserved in each step. However, this is not explicitly necessary for S-IFPROM, as both branches of a conditional with promoted discriminee already have $\top$ labels, which is enforced by the typing rules (e.g., the side condition in T-IFNODEP). S-PMATCHPROM and S-MATCHPROM are similar: the expressions these rules step to do not need to be promoted again, although their pattern variables are substituted by promoted values.

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

S-Ctx
$$\frac{\mathtt{e} \longrightarrow \mathtt{e}'}{\mathcal{E}[\mathtt{e}] \longrightarrow \mathcal{E}[\mathtt{e}']}$$

S-App
$$\frac{}{(\lambda \mathtt{x}:_l \tau \Rightarrow \mathtt{e})\ \mathtt{v} \longrightarrow [\mathtt{v}/\mathtt{x}]\mathtt{e}}$$

S-Let
$$\frac{}{\mathtt{let}\ \mathtt{x}:_l \tau = \mathtt{v}\ \mathtt{in}\ \mathtt{e} \longrightarrow [\mathtt{v}/\mathtt{x}]\mathtt{e}}$$

S-If
$$\frac{}{\mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \longrightarrow \mathtt{ite}(\mathtt{b},\mathtt{e}_1,\mathtt{e}_2)}$$

S-PMatch
$$\frac{}{\mathtt{match}\ (\mathtt{v}_1,\mathtt{v}_2)\ \mathtt{with}\ (\mathtt{x}_1,\mathtt{x}_2)\Rightarrow\mathtt{e} \longrightarrow [\mathtt{v}_2/\mathtt{x}_2][\mathtt{v}_1/\mathtt{x}_1]\mathtt{e}}$$

S-Match
$$\frac{}{\mathtt{match}\ \mathtt{C}_i\ \mathtt{v}\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x}\Rightarrow\mathtt{e}} \longrightarrow [\mathtt{v}/\mathtt{x}]\mathtt{e}_i}$$

S-OADT
$$\frac{\mathtt{obliv}\ \widehat{\mathtt{T}}\ (\mathtt{x}:\tau) = \tau' \in \Sigma}{\widehat{\mathtt{T}}\ \mathtt{v} \longrightarrow [\mathtt{v}/\mathtt{x}]\tau'}$$

S-Fun
$$\frac{\mathtt{fn}\ \mathtt{x}:_l \tau = \mathtt{e} \in \Sigma}{\mathtt{x} \longrightarrow \mathtt{e}}$$

S-SecBool
$$\frac{}{\widehat{\mathbb{B}}\#\mathtt{s}\ \mathtt{b} \longrightarrow [\mathtt{b}]}$$

S-SecInt
$$\frac{}{\widehat{\mathbb{Z}}\#\mathtt{s}\ \mathtt{n} \longrightarrow [\mathtt{n}]}$$

S-Add
$$\frac{}{\mathtt{n}_1 \oplus \mathtt{n}_2 \longrightarrow [\![\mathtt{n}_1 \oplus \mathtt{n}_2]\!]}$$

S-OAdd
$$\frac{}{[\mathtt{n}_1]\ \widehat{\oplus}\ [\mathtt{n}_2] \longrightarrow [[\![\mathtt{n}_1 \oplus \mathtt{n}_2]\!]]}$$

S-OInj
$$\frac{}{\widehat{\iota_b}\langle\widehat{\omega}\rangle\ \hat{\mathtt{v}} \longrightarrow [\iota_b\langle\widehat{\omega}\rangle\ \hat{\mathtt{v}}]}$$

S-Mux
$$\frac{}{\mathtt{mux}\ [\mathtt{b}]\ \mathtt{v}_1\ \mathtt{v}_2 \longrightarrow \mathtt{ite}(\mathtt{b},\mathtt{v}_1,\mathtt{v}_2)}$$

S-OPMatch
$$\frac{}{\widehat{\mathtt{match}}\ [\mathtt{v}_1,\mathtt{v}_2]\ \mathtt{with}\ [\mathtt{x}_1,\mathtt{x}_2]\Rightarrow\mathtt{e} \longrightarrow [\mathtt{v}_2/\mathtt{x}_2][\mathtt{v}_1/\mathtt{x}_1]\mathtt{e}}$$

S-OIf
$$\frac{}{\widehat{\mathcal{E}}[\widehat{\mathtt{if}}\ [\mathtt{b}]\ \mathtt{then}\ \mathtt{v}_1\ \mathtt{else}\ \mathtt{v}_2] \longrightarrow \widehat{\mathtt{if}}\ [\mathtt{b}]\ \mathtt{then}\ \widehat{\mathcal{E}}[\mathtt{v}_1]\ \mathtt{else}\ \widehat{\mathcal{E}}[\mathtt{v}_2]}$$

S-OMatch
$$\frac{\hat{\mathtt{v}}_1 \Leftarrow \widehat{\omega}_1 \qquad \hat{\mathtt{v}}_2 \Leftarrow \widehat{\omega}_2}{\widehat{\mathtt{match}}\ [\iota_b\langle\widehat{\omega}_1\ \widehat{+}\ \widehat{\omega}_2\rangle\ \hat{\mathtt{v}}]\ \mathtt{with}\ \mathtt{x}\Rightarrow\mathtt{e}_1 \mid \mathtt{x}\Rightarrow\mathtt{e}_2 \longrightarrow \begin{array}{l}\widehat{\mathtt{if}}\ [\mathtt{b}]\ \mathtt{then}\ \mathtt{ite}(\mathtt{b},[\hat{\mathtt{v}}/\mathtt{x}]\mathtt{e}_1,[\hat{\mathtt{v}}_1/\mathtt{x}]\mathtt{e}_1)\\ \mathtt{else}\ \mathtt{ite}(\mathtt{b},[\hat{\mathtt{v}}_2/\mathtt{x}]\mathtt{e}_2,[\hat{\mathtt{v}}/\mathtt{x}]\mathtt{e}_2)\end{array}}$$

Figure A.2. Core Taype semantics rules

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

S-TapeOIf

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

S-TapeProm

$$\mathtt{tape}\ (\uparrow \mathtt{v}) \longrightarrow \mathtt{v}$$

S-RetAdd$_1$

$$(\widehat{\mathbb{Z}}\#\mathtt{r}\ [\mathtt{n}_1]) \oplus (\widehat{\mathbb{Z}}\#\mathtt{r}\ [\mathtt{n}_2]) \longrightarrow \widehat{\mathbb{Z}}\#\mathtt{r}\ ([\mathtt{n}_1] \mathbin{\widehat{\oplus}} [\mathtt{n}_2])$$

S-RetAdd$_2$

$$(\widehat{\mathbb{Z}}\#\mathtt{r}\ [\mathtt{n}_1]) \oplus (\uparrow \mathtt{n}_2) \longrightarrow \widehat{\mathbb{Z}}\#\mathtt{r}\ ([\mathtt{n}_1] \mathbin{\widehat{\oplus}} (\widehat{\mathbb{Z}}\#\mathtt{s}\ \mathtt{n}_2))$$

S-RetAdd$_3$

$$(\uparrow \mathtt{n}_1) \oplus (\widehat{\mathbb{Z}}\#\mathtt{r}\ [\mathtt{n}_2]) \longrightarrow \widehat{\mathbb{Z}}\#\mathtt{r}\ ((\widehat{\mathbb{Z}}\#\mathtt{s}\ \mathtt{n}_1) \mathbin{\widehat{\oplus}} [\mathtt{n}_2])$$

S-SecRetInt

$$\widehat{\mathbb{Z}}\#\mathtt{s}\ (\widehat{\mathbb{Z}}\#\mathtt{r}\ [\mathtt{n}]) \longrightarrow \uparrow [\mathtt{n}]$$

S-SecIntProm

$$\widehat{\mathbb{Z}}\#\mathtt{s}\ (\uparrow \mathtt{n}) \longrightarrow \uparrow (\widehat{\mathbb{Z}}\#\mathtt{s}\ \mathtt{n})$$

S-SecBoolProm

$$\widehat{\mathbb{B}}\#\mathtt{s}\ (\uparrow \mathtt{b}) \longrightarrow \uparrow (\widehat{\mathbb{B}}\#\mathtt{s}\ \mathtt{b})$$

S-AddProm

$$(\uparrow \mathtt{n}_1) \oplus (\uparrow \mathtt{n}_2) \longrightarrow \uparrow (\mathtt{n}_1 \oplus \mathtt{n}_2)$$

S-AppProm

$$(\uparrow (\lambda \mathtt{x}:_\iota \tau \Rightarrow \mathtt{e}))\ \mathtt{v} \longrightarrow \uparrow ([\mathtt{v}/\mathtt{x}]\mathtt{e})$$

S-IfProm

$$\mathtt{if}\ \uparrow \mathtt{b}\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \longrightarrow \mathtt{ite}(\mathtt{b}, \mathtt{e}_1, \mathtt{e}_2)$$

S-PMatchProm

$$\mathtt{match}\ \uparrow (\mathtt{v}_1, \mathtt{v}_2)\ \mathtt{with}\ (\mathtt{x}_1, \mathtt{x}_2) \Rightarrow \mathtt{e} \longrightarrow [\uparrow \mathtt{v}_2/\mathtt{x}_2][\uparrow \mathtt{v}_1/\mathtt{x}_1]\mathtt{e}$$

S-MatchProm

$$\mathtt{match}\ \uparrow \mathtt{C}_i\ \mathtt{v}\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{e}} \longrightarrow [\uparrow \mathtt{v}/\mathtt{x}]\mathtt{e}_i$$

Figure A.3. Core Taype semantics rules (cont.)

A.2 Type System

Programs in TAYPE are typed using a pair of typing and kinding judgments: $\Sigma; \Gamma \vdash \mathtt{e} :_\iota \tau$ and $\Sigma; \Gamma \vdash \tau :: \kappa$. The global context Σ is elided for brevity, as it is fixed in all typing and kinding rules. Figure A.4 shows the kinding rules, while Figure A.5 and Figure A.6 present the typing rules.

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

$$\textsc{K-Sub}\ \frac{\Gamma \vdash \tau :: \kappa \qquad \kappa \sqsubseteq \kappa'}{\Gamma \vdash \tau :: \kappa'} \qquad \textsc{K-ADT}\ \frac{\mathtt{data\ T = \overline{C\ \tau} \in \Sigma}}{\Gamma \vdash \mathtt{T} :: *^{\mathsf{P}}} \qquad \textsc{K-OADT}\ \frac{\mathtt{obliv\ \widehat{T}\ (x{:}\tau) = \tau' \in \Sigma} \qquad \Gamma \vdash \mathtt{e} :_\perp \tau}{\Gamma \vdash \mathtt{\widehat{T}\ e} :: *^{\mathsf{O}}}$$

$$\textsc{K-Unit}\ \frac{}{\Gamma \vdash \mathbb{1} :: *^{\mathsf{A}}} \qquad \textsc{K-Bool}\ \frac{}{\Gamma \vdash \mathbb{B} :: *^{\mathsf{P}}} \qquad \textsc{K-OBool}\ \frac{}{\Gamma \vdash \widehat{\mathbb{B}} :: *^{\mathsf{O}}} \qquad \textsc{K-Int}\ \frac{}{\Gamma \vdash \mathbb{Z} :: *^{\mathsf{P}}} \qquad \textsc{K-OInt}\ \frac{}{\Gamma \vdash \widehat{\mathbb{Z}} :: *^{\mathsf{O}}}$$

$$\textsc{K-Pi}\ \frac{\Gamma \vdash \tau_1 :: * \qquad \mathtt{x} :_\iota \tau_1, \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \Pi \mathtt{x}{:}_\iota \tau_1, \tau_2 :: *^{\mathsf{M}}} \qquad \textsc{K-Prod}\ \frac{\Gamma \vdash \tau_1 :: \kappa \qquad \Gamma \vdash \tau_2 :: \kappa}{\Gamma \vdash \tau_1 \times \tau_2 :: \kappa \sqcup *^{\mathsf{P}}}$$

$$\textsc{K-OProd}\ \frac{\Gamma \vdash \tau_1 :: *^{\mathsf{O}} \qquad \Gamma \vdash \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash \tau_1 \mathbin{\widehat{\times}} \tau_2 :: *^{\mathsf{O}}} \qquad \textsc{K-OSum}\ \frac{\Gamma \vdash \tau_1 :: *^{\mathsf{O}} \qquad \Gamma \vdash \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash \tau_1 \mathbin{\widehat{+}} \tau_2 :: *^{\mathsf{O}}}$$

$$\textsc{K-Let}\ \frac{\Gamma \vdash \mathtt{e} :_\perp \tau \qquad \mathtt{x} :_\perp \tau, \Gamma \vdash \tau' :: *^{\mathsf{O}}}{\Gamma \vdash \mathtt{let\ x}{:}_\perp \tau\ \mathtt{=\ e\ in}\ \tau' :: *^{\mathsf{O}}} \qquad \textsc{K-If}\ \frac{\Gamma \vdash \mathtt{e}_0 :_\perp \mathbb{B} \qquad \Gamma \vdash \tau_1 :: *^{\mathsf{O}} \qquad \Gamma \vdash \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash \mathtt{if\ e}_0\ \mathtt{then}\ \tau_1\ \mathtt{else}\ \tau_2 :: *^{\mathsf{O}}}$$

$$\textsc{K-PMatch}\ \frac{\Gamma \vdash \mathtt{e}_0 :_\perp \tau_1 \times \tau_2 \qquad \mathtt{x}_1 :_\perp \tau_1, \mathtt{x}_2 :_\perp \tau_2, \Gamma \vdash \tau :: *^{\mathsf{O}}}{\Gamma \vdash \mathtt{match\ e}_0\ \mathtt{with\ (x}_1\mathtt{,x}_2\mathtt{)} \Rightarrow \tau :: *^{\mathsf{O}}}$$

$$\textsc{K-Match}\ \frac{\mathtt{data\ T = \overline{C\ \tau} \in \Sigma} \qquad \Gamma \vdash \mathtt{e}_0 :_\perp \mathtt{T} \qquad \forall i.\ \mathtt{x} :_\perp \tau_i, \Gamma \vdash \tau_i' :: *^{\mathsf{O}}}{\Gamma \vdash \mathtt{match\ e}_0\ \mathtt{with}\ \overline{\mathtt{C\ x} \Rightarrow \tau'} :: *^{\mathsf{O}}}$$

Figure A.4. Core TAYPE kinding rules

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

T-Conv
$$\frac{\Gamma \vdash \mathtt{e} :_l \tau \qquad \tau \equiv \tau' \qquad \Gamma \vdash \tau' :: *}{\Gamma \vdash \mathtt{e} :_l \tau'}$$

T-Var
$$\frac{\mathtt{x} :_l \tau \in \Gamma}{\Gamma \vdash \mathtt{x} :_l \tau}$$

T-Unit
$$\frac{}{\Gamma \vdash () :_\bot \mathbb{1}}$$

T-LitBool
$$\frac{}{\Gamma \vdash \mathtt{b} :_\bot \mathbb{B}}$$

T-LitInt
$$\frac{}{\Gamma \vdash \mathtt{n} :_\bot \mathbb{Z}}$$

T-Abs
$$\frac{\mathtt{x} :_{l_1} \tau_1, \Gamma \vdash \mathtt{e} :_{l_2} \tau_2 \qquad \Gamma \vdash \tau_1 :: *}{\Gamma \vdash \lambda \mathtt{x}{:}_{l_1}\tau_1 \Rightarrow \mathtt{e} :_{l_2} \Pi \mathtt{x}{:}_{l_1}\tau_1, \tau_2}$$

T-App
$$\frac{\Gamma \vdash \mathtt{e}_2 :_{l_2} \Pi \mathtt{x}{:}_{l_1}\tau_1, \tau_2 \qquad \Gamma \vdash \mathtt{e}_1 :_{l_1} \tau_1}{\Gamma \vdash \mathtt{e}_2\ \mathtt{e}_1 :_{l_2} [\mathtt{e}_1/\mathtt{x}]\tau_2}$$

T-Let
$$\frac{\Gamma \vdash \mathtt{e}_1 :_{l_1} \tau_1 \qquad \mathtt{x} :_{l_1} \tau_1, \Gamma \vdash \mathtt{e}_2 :_{l_2} \tau_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{:}_{l_1}\tau_1\ =\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 :_{l_2} [\mathtt{e}_1/\mathtt{x}]\tau_2}$$

T-Fun
$$\frac{\mathtt{fn}\ \mathtt{x}{:}_l\tau\ =\ \mathtt{e} \in \Sigma}{\Gamma \vdash \mathtt{x} :_l \tau}$$

T-If
$$\frac{\Gamma \vdash \mathtt{e}_0 :_\bot \mathbb{B} \qquad \Gamma \vdash \mathtt{e}_1 :_l [\mathtt{true}/\mathtt{z}]\tau \qquad \Gamma \vdash \mathtt{e}_2 :_l [\mathtt{false}/\mathtt{z}]\tau}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 :_l [\mathtt{e}_0/\mathtt{z}]\tau}$$

T-IfNoDep
$$\frac{\Gamma \vdash \mathtt{e}_0 :_{l_0} \mathbb{B} \qquad l_0 \sqsubseteq l \qquad \Gamma \vdash \mathtt{e}_1 :_l \tau \qquad \Gamma \vdash \mathtt{e}_2 :_l \tau}{\Gamma \vdash \mathtt{if}_\tau\ \mathtt{e}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 :_l \tau}$$

T-Pair
$$\frac{\Gamma \vdash \mathtt{e}_1 :_l \tau_1 \qquad \Gamma \vdash \mathtt{e}_2 :_l \tau_2}{\Gamma \vdash (\mathtt{e}_1, \mathtt{e}_2) :_l \tau_1 \times \tau_2}$$

T-PMatch
$$\frac{\Gamma \vdash \mathtt{e}_0 :_\bot \tau_1 \times \tau_2 \qquad \mathtt{x}_1 :_\bot \tau_1, \mathtt{x}_2 :_\bot \tau_2, \Gamma \vdash \mathtt{e} :_l [(\mathtt{x}_1, \mathtt{x}_2)/\mathtt{z}]\tau}{\Gamma \vdash \mathtt{match}\ \mathtt{e}_0\ \mathtt{with}\ (\mathtt{x}_1, \mathtt{x}_2) \Rightarrow \mathtt{e} :_l [\mathtt{e}_0/\mathtt{z}]\tau}$$

T-PMatchNoDep
$$\frac{\Gamma \vdash \mathtt{e}_0 :_{l_0} \tau_1 \times \tau_2 \qquad l_0 \sqsubseteq l \qquad \mathtt{x}_1 :_{l_0} \tau_1, \mathtt{x}_2 :_{l_0} \tau_2, \Gamma \vdash \mathtt{e} :_l \tau}{\Gamma \vdash \mathtt{match}_\tau\ \mathtt{e}_0\ \mathtt{with}\ (\mathtt{x}_1, \mathtt{x}_2) \Rightarrow \mathtt{e} :_l \tau}$$

T-Add
$$\frac{\Gamma \vdash \mathtt{e}_1 :_l \mathbb{Z} \qquad \Gamma \vdash \mathtt{e}_2 :_l \mathbb{Z}}{\Gamma \vdash \mathtt{e}_1 \oplus \mathtt{e}_2 :_l \mathbb{Z}}$$

T-Ctor
$$\frac{\mathtt{data}\ \mathtt{T}\ =\ \overline{\mathtt{C}\ \tau} \in \Sigma \qquad \Gamma \vdash \mathtt{e} :_l \tau_i}{\Gamma \vdash \mathtt{C}_i\ \mathtt{e} :_l \mathtt{T}}$$

T-Match
$$\frac{\mathtt{data}\ \mathtt{T}\ =\ \overline{\mathtt{C}\ \tau} \in \Sigma \qquad \Gamma \vdash \mathtt{e}_0 :_\bot \mathtt{T} \qquad \forall i.\ \mathtt{x} :_\bot \tau_i, \Gamma \vdash \mathtt{e}_i :_l [\mathtt{C}_i\ \mathtt{x}/\mathtt{z}]\tau'}{\Gamma \vdash \mathtt{match}\ \mathtt{e}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{e}} :_l [\mathtt{e}_0/\mathtt{z}]\tau'}$$

T-MatchNoDep
$$\frac{\mathtt{data}\ \mathtt{T}\ =\ \overline{\mathtt{C}\ \tau} \in \Sigma \qquad \Gamma \vdash \mathtt{e}_0 :_{l_0} \mathtt{T} \qquad l_0 \sqsubseteq l \qquad \forall i.\ \mathtt{x} :_{l_0} \tau_i, \Gamma \vdash \mathtt{e}_i :_l \tau'}{\Gamma \vdash \mathtt{match}_{\tau'}\ \mathtt{e}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{e}} :_l \tau'}$$

Figure A.5. Core Taype typing rules

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

T-Mux
$$\frac{\Gamma \vdash \mathsf{e}_0 :_\bot \widehat{\mathbb{B}} \qquad \Gamma \vdash \tau :: *^{\mathsf{O}} \qquad \Gamma \vdash \mathsf{e}_1 :_\bot \tau \qquad \Gamma \vdash \mathsf{e}_2 :_\bot \tau}{\Gamma \vdash \texttt{mux}\ \mathsf{e}_0\ \mathsf{e}_1\ \mathsf{e}_2 :_\bot \tau}$$

T-OPair
$$\frac{\Gamma \vdash \mathsf{e}_1 :_\bot \tau_1 \qquad \Gamma \vdash \mathsf{e}_2 :_\bot \tau_2 \qquad \Gamma \vdash \tau_1 :: *^{\mathsf{O}} \qquad \Gamma \vdash \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash [\mathsf{e}_1, \mathsf{e}_2] :_\bot \tau_1 \widehat{\times} \tau_2}$$

T-OPMatch
$$\frac{\Gamma \vdash \mathsf{e}_0 :_\bot \tau_1 \widehat{\times} \tau_2 \qquad \mathsf{x}_1 :_\bot \tau_1, \mathsf{x}_2 :_\bot \tau_2, \Gamma \vdash \mathsf{e} :_l \tau}{\Gamma \vdash \widehat{\texttt{match}}\ \mathsf{e}_0 : \tau_1 \widehat{\times} \tau_2\ \texttt{with}\ [\mathsf{x}_1, \mathsf{x}_2] \Rightarrow \mathsf{e} :_l \tau}$$

T-OAdd
$$\frac{\Gamma \vdash \mathsf{e}_1 :_\bot \widehat{\mathbb{Z}} \qquad \Gamma \vdash \mathsf{e}_2 :_\bot \widehat{\mathbb{Z}}}{\Gamma \vdash \mathsf{e}_1 \widehat{\oplus} \mathsf{e}_2 :_\bot \widehat{\mathbb{Z}}}$$

T-OInj
$$\frac{\Gamma \vdash \mathsf{e} :_\bot \texttt{ite}(\mathsf{b}, \tau_1, \tau_2) \qquad \Gamma \vdash \tau_1 \widehat{+} \tau_2 :: *^{\mathsf{O}}}{\Gamma \vdash \widehat{\iota_b}\langle \tau_1 \widehat{+} \tau_2 \rangle\ \mathsf{e} :_\bot \tau_1 \widehat{+} \tau_2}$$

T-OMatch
$$\frac{\Gamma \vdash \mathsf{e}_0 :_\bot \tau_1 \widehat{+} \tau_2 \qquad \mathsf{x} :_\bot \tau_1, \Gamma \vdash \mathsf{e}_1 :_\top \tau \qquad \mathsf{x} :_\bot \tau_2, \Gamma \vdash \mathsf{e}_2 :_\top \tau}{\Gamma \vdash \widehat{\texttt{match}}_\tau\ \mathsf{e}_0 : \tau_1 \widehat{+} \tau_2\ \texttt{with}\ \mathsf{x} \Rightarrow \mathsf{e}_1 \mid \mathsf{x} \Rightarrow \mathsf{e}_2 :_\top \tau}$$

T-BoxedLitBool
$$\frac{}{\Gamma \vdash [\mathsf{b}] :_\bot \widehat{\mathbb{B}}}$$

T-BoxedLitInt
$$\frac{}{\Gamma \vdash [\mathsf{n}] :_\bot \widehat{\mathbb{Z}}}$$

T-BoxedInj
$$\frac{[\iota_b \langle \widehat{\omega} \rangle\ \hat{\mathsf{v}}] \Leftarrow \widehat{\omega}}{\Gamma \vdash [\iota_b \langle \widehat{\omega} \rangle\ \hat{\mathsf{v}}] :_\bot \widehat{\omega}}$$

T-SecBool
$$\frac{\Gamma \vdash \mathsf{e} :_l \mathbb{B}}{\Gamma \vdash \widehat{\mathbb{B}} \texttt{\#s}\ \mathsf{e} :_l \widehat{\mathbb{B}}}$$

T-SecInt
$$\frac{\Gamma \vdash \mathsf{e} :_l \mathbb{Z}}{\Gamma \vdash \widehat{\mathbb{Z}} \texttt{\#s}\ \mathsf{e} :_l \widehat{\mathbb{Z}}}$$

T-RetInt
$$\frac{\Gamma \vdash \mathsf{e} :_\bot \widehat{\mathbb{Z}}}{\Gamma \vdash \widehat{\mathbb{Z}} \texttt{\#r}\ \mathsf{e} :_\top \mathbb{Z}}$$

T-OIf
$$\frac{\Gamma \vdash \mathsf{e}_0 :_\bot \widehat{\mathbb{B}} \qquad \Gamma \vdash \mathsf{e}_1 :_\top \tau \qquad \Gamma \vdash \mathsf{e}_2 :_\top \tau}{\Gamma \vdash \widehat{\texttt{if}}\ \mathsf{e}_0\ \texttt{then}\ \mathsf{e}_1\ \texttt{else}\ \mathsf{e}_2 :_\top \tau}$$

T-Promote
$$\frac{\Gamma \vdash \mathsf{e} :_\bot \tau}{\Gamma \vdash \uparrow \mathsf{e} :_\top \tau}$$

T-Tape
$$\frac{\Gamma \vdash \mathsf{e} :_\top \tau \qquad \Gamma \vdash \tau :: *^{\mathsf{O}}}{\Gamma \vdash \texttt{tape}\ \mathsf{e} :_\bot \tau}$$

Figure A.6. Core Taype typing rules (cont.)

A.3 Translation Algorithm

Figure A.7 translates TAYPE oblivious types to OIL expressions of size type. Figure A.8 and Figure A.9 translate TAYPE expressions to OIL expressions.

$\boxed{\Gamma \vdash \tau \rightsquigarrow \mathtt{s}}$

$$\textsc{TR-UnitT}\ \frac{}{\Gamma \vdash \mathbb{1} \rightsquigarrow \mathtt{0}} \qquad \textsc{TR-OBool}\ \frac{}{\Gamma \vdash \widehat{\mathbb{B}} \rightsquigarrow \mathtt{1}} \qquad \textsc{TR-OInt}\ \frac{}{\Gamma \vdash \widehat{\mathbb{Z}} \rightsquigarrow \mathtt{1}}$$

$$\textsc{TR-OProd}\ \frac{\Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \tau_1 \mathbin{\widehat{\times}} \tau_2 \rightsquigarrow \mathtt{s}_1 + \mathtt{s}_2}$$

$$\textsc{TR-OSum}\ \frac{\Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \tau_1 \mathbin{\widehat{+}} \tau_2 \rightsquigarrow \mathtt{1} + \mathtt{max}\ \mathtt{s}_1\ \mathtt{s}_2} \qquad \textsc{TR-TApp}\ \frac{}{\Gamma \vdash \widehat{\mathtt{T}}\ \mathtt{x} \rightsquigarrow \widehat{\mathtt{T}}\ \mathtt{x}}$$

$$\textsc{TR-TLet}\ \frac{\Gamma \vdash \mathtt{e} \rightsquigarrow_\perp \dot{\mathtt{e}} \qquad \mathtt{x} :_\perp \tau_1, \Gamma \vdash \tau \rightsquigarrow \mathtt{s}}{\Gamma \vdash \mathtt{let}\ \mathtt{x} :_\perp \tau_1\ =\ \mathtt{e}\ \mathtt{in}\ \tau \rightsquigarrow \mathtt{let}\ \mathtt{x}\ =\ \dot{\mathtt{e}}\ \mathtt{in}\ \mathtt{s}}$$

$$\textsc{TR-TIf}\ \frac{\Gamma \vdash \tau_1 \rightsquigarrow \mathtt{s}_1 \qquad \Gamma \vdash \tau_2 \rightsquigarrow \mathtt{s}_2}{\Gamma \vdash \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \tau_1\ \mathtt{else}\ \tau_2 \rightsquigarrow \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{s}_1\ \mathtt{else}\ \mathtt{s}_2}$$

$$\textsc{TR-TPMatch}\ \frac{\mathtt{x}_0 :_\perp \tau_1 \times \tau_2 \in \Gamma \qquad \mathtt{x}_1 :_\perp \tau_1, \mathtt{x}_2 :_\perp \tau_2, \Gamma \vdash \tau \rightsquigarrow \mathtt{s}}{\Gamma \vdash \mathtt{match}\ \mathtt{x}_0\ \mathtt{with}\ (\mathtt{x}_1, \mathtt{x}_2) \Rightarrow \tau \rightsquigarrow \mathtt{match}\ \mathtt{x}_0\ \mathtt{with}\ (\mathtt{x}_1, \mathtt{x}_2) \Rightarrow \mathtt{s}}$$

$$\textsc{TR-TMatch}\ \frac{\mathtt{data}\ \mathtt{T} = \overline{\mathtt{C}\ \tau} \in \Sigma \qquad \mathtt{x}_0 :_\perp \mathtt{T} \in \Gamma \qquad \forall i.\ \mathtt{x} :_\perp \tau_i, \Gamma \vdash \tau_i' \rightsquigarrow \mathtt{s}_i}{\Gamma \vdash \mathtt{match}\ \mathtt{x}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \tau'} \rightsquigarrow \mathtt{match}\ \mathtt{x}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{s}}}$$

Figure A.7. Translating core TAYPE oblivious types to OIL sizes

$\boxed{\Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}}$

TR-Bool
$$\frac{}{\Gamma \vdash \mathtt{b} \rightsquigarrow_\bot \mathtt{b}}$$

TR-Int
$$\frac{}{\Gamma \vdash \mathtt{n} \rightsquigarrow_\bot \mathtt{n}}$$

TR-Var
$$\frac{}{\Gamma \vdash \mathtt{x} \rightsquigarrow_l \mathtt{x}}$$

TR-Abs
$$\frac{\mathtt{x} :_{l_1} \tau_1, \Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}}{\Gamma \vdash \lambda \mathtt{x}{:}_{l_1}\tau_1 \Rightarrow \mathtt{e} \rightsquigarrow_l \lambda \mathtt{x} \Rightarrow \dot{\mathtt{e}}}$$

TR-App
$$\frac{}{\Gamma \vdash \mathtt{x}_2\ \mathtt{x}_1 \rightsquigarrow_l \mathtt{x}_2\ \mathtt{x}_1}$$

TR-Pair
$$\frac{}{\Gamma \vdash (\mathtt{x}_1,\mathtt{x}_2) \rightsquigarrow_l \begin{cases} (\mathtt{x}_1,\mathtt{x}_2) & \text{if } l = \bot \\ \widetilde{\mathtt{pair}}\ \mathtt{x}_1\ \mathtt{x}_2 & \text{if } l = \top \end{cases}}$$

TR-Ctor
$$\frac{}{\Gamma \vdash \mathtt{C}\ \mathtt{x} \rightsquigarrow_l \begin{cases} \mathtt{C}\ \mathtt{x} & \text{if } l_0 = \bot \\ \widetilde{\mathtt{C}}\ \mathtt{x} & \text{if } l_0 = \top \end{cases}}$$

TR-Add
$$\frac{}{\Gamma \vdash \mathtt{x}_1 \oplus \mathtt{x}_2 \rightsquigarrow_l \begin{cases} \mathtt{x}_1 \oplus \mathtt{x}_2 & \text{if } l_0 = \bot \\ \mathtt{x}_1 \widetilde{\oplus} \mathtt{x}_2 & \text{if } l_0 = \top \end{cases}}$$

TR-SecBool
$$\frac{}{\Gamma \vdash \widehat{\mathbb{B}}\#\mathtt{s}\ \mathtt{x} \rightsquigarrow_l \begin{cases} \widehat{\mathbb{B}}\#\mathtt{s}\ \mathtt{x} & \text{if } l = \bot \\ \widetilde{\mathbb{B}\#\mathtt{s}}\ \mathtt{x} & \text{if } l = \top \end{cases}}$$

TR-SecInt
$$\frac{}{\Gamma \vdash \widehat{\mathbb{Z}}\#\mathtt{s}\ \mathtt{x} \rightsquigarrow_l \begin{cases} \widehat{\mathbb{Z}}\#\mathtt{s}\ \mathtt{x} & \text{if } l = \bot \\ \widetilde{\mathbb{Z}\#\mathtt{s}}\ \mathtt{x} & \text{if } l = \top \end{cases}}$$

TR-Let
$$\frac{\Gamma \vdash \mathtt{e}_1 \rightsquigarrow_{l_1} \dot{\mathtt{e}}_1 \qquad \mathtt{x} :_{l_1} \tau_1, \Gamma \vdash \mathtt{e}_2 \rightsquigarrow_l \dot{\mathtt{e}}_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{:}_{l_1}\tau_1\ =\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \rightsquigarrow_l \mathtt{let}\ \mathtt{x}\ =\ \dot{\mathtt{e}}_1\ \mathtt{in}\ \dot{\mathtt{e}}_2}$$

TR-If
$$\frac{\mathtt{x}_0 :_{l_0} \mathbb{B} \in \Gamma \qquad \Gamma \vdash \mathtt{e}_1 \rightsquigarrow_l \dot{\mathtt{e}}_1 \qquad \Gamma \vdash \mathtt{e}_2 \rightsquigarrow_l \dot{\mathtt{e}}_2}{\Gamma \vdash \mathtt{if}_\tau\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 \rightsquigarrow_l \begin{cases} \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \dot{\mathtt{e}}_1\ \mathtt{else}\ \dot{\mathtt{e}}_2 & \text{if } l_0 = \bot \\ \widetilde{\mathtt{if}}\ \widehat{\mathtt{if}}(\tau)\ \mathtt{x}_0\ \dot{\mathtt{e}}_1\ \dot{\mathtt{e}}_2 & \text{if } l_0 = \top \end{cases}}$$

TR-PMatch
$$\frac{\mathtt{x}_0 :_{l_0} \tau_1 \times \tau_2 \in \Gamma \qquad \mathtt{x}_1 :_{l_0} \tau_1, \mathtt{x}_2 :_{l_0} \tau_2, \Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}}{\begin{array}{l}\Gamma \vdash \mathtt{match}_\tau\ \mathtt{x}_0\ \mathtt{with}\ (\mathtt{x}_1,\mathtt{x}_2) \Rightarrow \mathtt{e} \\ \qquad \rightsquigarrow_l \begin{cases} \mathtt{match}\ \mathtt{x}_0\ \mathtt{with}\ (\mathtt{x}_1,\mathtt{x}_2) \Rightarrow \dot{\mathtt{e}} & \text{if } l_0 = \bot \\ \widetilde{\mathtt{match}}_\times\ \mathtt{prom}(\tau_1)\ \mathtt{prom}(\tau_2)\ \widehat{\mathtt{if}}(\tau)\ \mathtt{x}_0\ (\lambda \mathtt{x}_1 \lambda \mathtt{x}_2 \Rightarrow \dot{\mathtt{e}}) & \text{if } l_0 = \top \end{cases}\end{array}}$$

TR-Match
$$\frac{\mathtt{data}\ \mathtt{T}\ =\ \overline{\mathtt{C}\ \tau} \in \Sigma \qquad \mathtt{x}_0 :_{l_0} \mathtt{T} \in \Gamma \qquad \forall i.\ \mathtt{x} :_{l_0} \tau_i, \Gamma \vdash \mathtt{e}_i \rightsquigarrow_l \dot{\mathtt{e}}_i}{\Gamma \vdash \mathtt{match}_\tau\ \mathtt{x}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{e}} \rightsquigarrow_l \begin{cases} \mathtt{match}\ \mathtt{x}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \dot{\mathtt{e}}} & \text{if } l_0 = \bot \\ \widetilde{\mathtt{match}}_\mathtt{T}\ \widehat{\mathtt{if}}(\tau)\ \mathtt{x}_0\ \overline{(\lambda \mathtt{x} \Rightarrow \dot{\mathtt{e}})} & \text{if } l_0 = \top \end{cases}}$$

Figure A.8. Translating core Taype expressions to Oil expressions

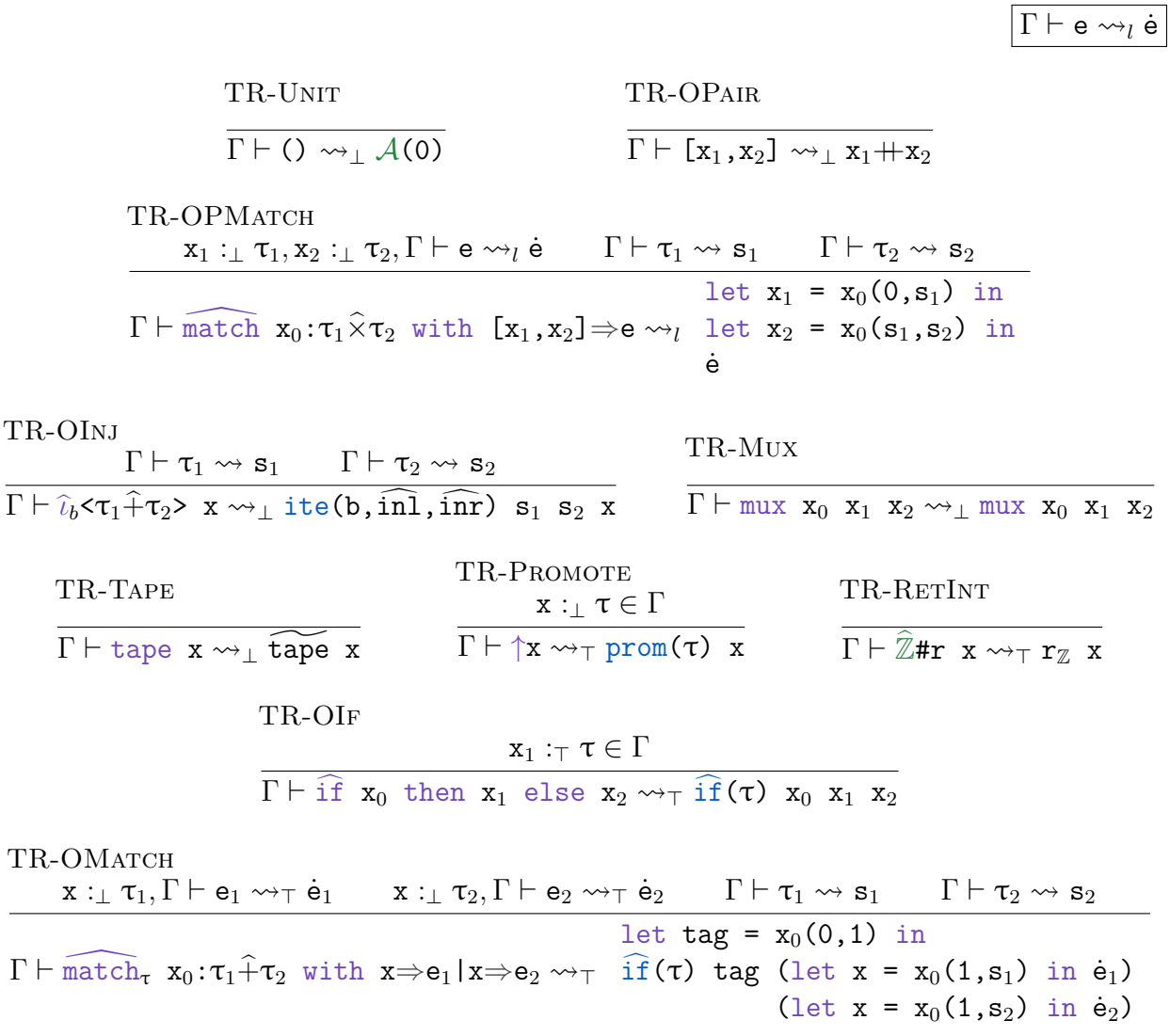


Figure A.9. Translating core Taype expressions to Oil expressions (cont.)

A.4 Totality of the Translation Algorithm

The totality theorem relies on a few lemmas that we state as follows.

Lemma A.4.1 (Regularity)**.** *If* $\Gamma \vdash \mathtt{e} :_l \tau$*, then* $\Gamma \vdash \tau :: \kappa$ *for some kind* κ*.*

Lemma A.4.2 (Conversion of typing context)**.** *If* $\mathtt{x} :_{l'} \tau_1, \Gamma \vdash \mathtt{e} :_l \tau$ *and* $\tau_1 \equiv \tau_2$ *with* $\Gamma \vdash \tau_2 :: *$*, then* $\mathtt{x} :_{l'} \tau_2, \Gamma \vdash \mathtt{e} :_l \tau$*.*

If $\mathtt{x} :_{l'} \tau_1, \Gamma \vdash \tau :: \kappa$ *and* $\tau_1 \equiv \tau_2$ *with* $\Gamma \vdash \tau_2 :: *$*, then* $\mathtt{x} :_{l'} \tau_2, \Gamma \vdash \tau :: \kappa$*.*

Lemma A.4.3 (Equality of equivalent public types)**.**

1. *If* $\Gamma \vdash \tau :: *$ *and* $\tau \equiv \mathbb{B}$*, then* $\tau = \mathbb{B}$*.*
2. *If* $\Gamma \vdash \tau :: *$*,* $\Gamma \vdash \mathtt{T} :: *$ *and* $\tau \equiv \mathtt{T}$*, then* $\tau = \mathtt{T}$*.*
3. *If* $\Gamma \vdash \tau :: *$*,* $\Gamma \vdash \tau_1 \times \tau_2 :: *$ *and* $\tau \equiv \tau_1 \times \tau_2$*, then* $\tau = \tau_1' \times \tau_2'$ *for some* τ_1' *and* τ_2' *such that* $\tau_1 \equiv \tau_1'$ *and* $\tau_2 \equiv \tau_2'$

We will also use the expected typing and kinding inversion lemmas. The proofs of these lemmas are omitted here, which are available in the Coq formalization [49]. The last lemma essentially says if a well-kinded type is equivalent to a well-kinded public type, then they are equal. The case of product also falls into this interpretation if we consider the type former $\times$ itself public, even though $\times$ can be used to connect non-public components. In other words, the "heads" of the public type formers are equal.

Now we prove the following totality theorem.

Theorem A.4.4 (Totality of translation)**.** *If* $\Gamma \vdash \mathtt{e} :_l \tau$ *and* $\mathtt{e}$ *is in ANF, then* $\Gamma \vdash \mathtt{e} \rightsquigarrow_l \dot{\mathtt{e}}$ *for some* Oil *expression* $\dot{\mathtt{e}}$*.*

If $\Gamma \vdash \tau :: *^{\mathsf{O}}$ *and* τ *is in ANF, then* $\Gamma \vdash \tau \rightsquigarrow \mathtt{s}$ *for some* Oil *expression* $\mathtt{s}$*.*

Proof. By induction on the ANF structure of $\mathtt{e}$ and τ, we prove these two statements simultaneously. Every subterm of an ANF term is also in ANF, so we can always apply the induction hypotheses to subterms.

It is trivial to prove the cases on base oblivious types, literals, variable, constructor, (function and type) application, public and oblivious pairs, integer operations, primitive

sections and retractions, atomic conditional $\texttt{mux}$ and the $\texttt{tape}$ operation. We simply apply their corresponding translation rules, which do not have any assumptions.

The cases on public types (booleans, integers and public products) and function type vacuously hold, since they are not obliviously kinded.

Case on lambda abstraction $\lambda \texttt{x}:_{l_1}\tau_1 \Rightarrow \texttt{e}$: By the well-typedness assumption and the inversion lemma, we have $\texttt{x} :_{l_1} \tau_1, \Gamma \vdash \texttt{e} :_l \tau'$ for some τ'. It then follows that $\texttt{x} :_{l_1} \tau_1, \Gamma \vdash \texttt{e} \rightsquigarrow_l \dot{\texttt{e}}$ for some $\dot{\texttt{e}}$, by the induction hypothesis. Applying TR-Abs concludes this case.

Case on let binding $\texttt{let x}:_{l_1}\tau_1 \texttt{ = } \texttt{e}_1 \texttt{ in } \texttt{e}_2$: The proof for the first part (when this expression is well-typed) is similar to the case on lambda abstraction using rule TR-Let. On the other hand, if this is obliviously kinded, we have $\Gamma \vdash \texttt{e}_1 :_\perp \tau_1$ and $\texttt{x} :_\perp \tau_1, \Gamma \vdash \texttt{e}_2 :: *^{\mathsf{O}}$. By the induction hypotheses, $\Gamma \vdash \texttt{e}_1 \rightsquigarrow_\perp \dot{\texttt{e}}_1$ and $\texttt{x} :_\perp \tau_1, \Gamma \vdash \texttt{e}_2 \rightsquigarrow \texttt{s}$ for some $\dot{\texttt{e}}_1$ and $\texttt{s}$. We then discharge this case by TR-TLet.

Case on oblivious injection $\widehat{\imath}_b\texttt{<}\tau\texttt{>}\ \texttt{x}$: By the well-typedness assumption and the inversion lemma, we know $l = \perp$ and τ is some $\tau_1 \widehat{+} \tau_2$ such that $\Gamma \vdash \tau_1 \widehat{+} \tau_2 :: *^{\mathsf{O}}$, which implies that τ_1 and τ_2 are also obliviously kinded by the kinding inversion lemma. The conclusion follows by TR-OInj.

Case on promotion $\uparrow\texttt{x}$: We have $\Gamma \vdash \texttt{x} :_\perp \tau$ from the assumption. By the inversion lemma for variables, we get $\texttt{x} :_\perp \tau' \in \Gamma$ for some equivalent type τ'. This case then follows by TR-Promote.

Cases on oblivious product and sum types, their pattern matching expressions and leaky conditional are similar.

The remaining cases on conditional, product and ADT pattern matching are the trickiest ones. We show the proofs for conditional and product pattern matching here; the case of of ADT pattern matching is analogous.

In the case of well-typed conditional, $\texttt{if}_\tau\ \texttt{x}_0\ \texttt{then}\ \texttt{e}_1\ \texttt{else}\ \texttt{e}_2$, we know $\texttt{e}_1$ and $\texttt{e}_2$ are well-typed with some label $\texttt{l}$, which discharges the two corresponding conditions in TR-If by the induction hypotheses. By the inversion lemma for variables and the fact that $\Gamma \vdash \texttt{x}_0 :_{l_0} \mathbb{B}$, we have $\texttt{x}_0 :_{l_0} \tau' \in \Gamma$, for some τ' such that $\tau' \equiv \mathbb{B}$ and τ' is well-kinded. However, τ' must equal to $\mathbb{B}$ by Lemma A.4.3, thus $\texttt{x}_0 :_{l_0} \mathbb{B} \in \Gamma$, required by TR-If. The case of obliviously kinded conditional is similar using TR-TIf.

In the case of well-typed product pattern matching, $\texttt{match}_{\tau}\ \texttt{x}_0\ \texttt{with}\ (\texttt{x}_1,\texttt{x}_2)\Rightarrow\texttt{e}$, we know $\Gamma \vdash \texttt{x}_0 :_{l_0} \tau_1' \times \tau_2'$, and $\texttt{e}$ is well-typed under the context $\texttt{x}_1 :_{l_0} \tau_1', \texttt{x}_2 :_{l_0} \tau_2', \Gamma$. By the inversion lemma for variables, $\texttt{x}_0 :_{l_0} \tau' \in \Gamma$ for some well-kinded τ' and $\tau' \equiv \tau_1' \times \tau_2'$. As $\tau_1' \times \tau_2'$ is well-kinded by Lemma A.4.1, it follows from Lemma A.4.3 that $\tau' = \tau_1 \times \tau_2$ for some τ_1 and τ_2, such that $\tau_1' \equiv \tau_1$ and $\tau_2' \equiv \tau_2$. We then conclude this case by applying TR-PMatch, whose translation premise is obtained by Lemma A.4.2 and the induction hypothesis. The case for obliviously kinded product pattern matching is similar. □

B. OMITTED RULES AND PROOFS FOR TAYPSI

This chapter documents the omitted rules from Chapter 6: full logical refinement definitions and declarative lifting rules (Appendix B.1), as well as full algorithmic lifting rules (Appendix B.2). Appendix B.3 proves the correctness of declarative lifting and the soundness of algorithmic lifting.

B.1 Declarative Lifting

Figure B.1 gives the full definitions of the logical refinement from Section 6.3.6. In particular, Figure B.1 defines an interpretation for typing contexts that range over specification types, $\mathcal{G}_n[\![\Gamma]\!]$. The codomain of this interpretation is substitutions (σ) of related value pairs.

$\boxed{\mathcal{V}_n[\![\theta]\!]}$

$$\mathcal{V}_n[\![\mathbb{1}]\!] = \mathcal{V}_n[\![\mathbb{B}]\!] = \mathcal{V}_n[\![\mathtt{T}]\!] = \{\, (\mathtt{v}, \mathtt{v}') \mid 0 < n \implies \mathtt{v} = \mathtt{v}' \,\}$$

$$\mathcal{V}_n[\![\widehat{\mathbb{B}}]\!] = \{\, (\mathtt{b}, \mathtt{[b']}) \mid 0 < n \implies \mathtt{b} = \mathtt{b}' \,\}$$

$$\mathcal{V}_n[\![\Psi\widehat{\mathtt{T}}]\!] = \{\, (\mathtt{v}, \langle \mathtt{k}, \widehat{\mathtt{v}} \rangle) \mid 0 < n \implies \mathtt{r\ k\ } \widehat{\mathtt{v}} \longrightarrow^* \mathtt{v} \,\}$$

$$\mathcal{V}_n[\![\theta_1 \times \theta_2]\!] = \{\, ((\mathtt{v}_1, \mathtt{v}_2), (\mathtt{v}_1', \mathtt{v}_2')) \mid (\mathtt{v}_1, \mathtt{v}_1') \in \mathcal{V}_n[\![\theta_1]\!] \wedge (\mathtt{v}_2, \mathtt{v}_2') \in \mathcal{V}_n[\![\theta_2]\!] \,\}$$

$$\mathcal{V}_n[\![\theta_1 \to \theta_2]\!] = \left\{ (\lambda \mathtt{x}{:}\lfloor\theta_1\rfloor \Rightarrow \mathtt{e}, \lambda \mathtt{x}{:}\theta_1 \Rightarrow \mathtt{e}') \;\middle|\; \begin{array}{r} \forall i < n.\ \forall (\mathtt{v}, \mathtt{v}') \in \mathcal{V}_i[\![\theta_1]\!]. \\ (\mathtt{[v/x]e}, \mathtt{[v'/x]e}') \in \mathcal{E}_i[\![\theta_2]\!] \end{array} \right\}$$

$\boxed{\mathcal{E}_n[\![\theta]\!]}$

$$\mathcal{E}_n[\![\theta]\!] = \left\{ (\mathtt{e}, \mathtt{e}') \;\middle|\; \forall i < n.\ \forall \mathtt{v}'.\ \mathtt{e}' \longrightarrow^i \mathtt{v}' \implies \exists \mathtt{v}.\ \mathtt{e} \longrightarrow^* \mathtt{v} \wedge (\mathtt{v}, \mathtt{v}') \in \mathcal{V}_{n-i}[\![\theta]\!] \right\}$$

$\boxed{\mathcal{G}_n[\![\Gamma]\!]}$

$$\mathcal{G}_n[\![\cdot]\!] = \{\, \emptyset \,\} \qquad \mathcal{G}_n[\![\mathtt{x} : \theta, \Gamma]\!] = \{\, \sigma[\mathtt{x} \mapsto (\mathtt{v}, \mathtt{v}')] \mid \sigma \in \mathcal{G}_n[\![\Gamma]\!] \wedge (\mathtt{v}, \mathtt{v}') \in \mathcal{V}_n[\![\theta]\!] \,\}$$

Figure B.1. Refinement as logical relation

Figure B.2 presents the full rules of the declarative lifting relation, $\mathcal{S}; \mathcal{L}; \Sigma; \Gamma \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}$. We elide most contexts as they are fixed in these rules for brevity.

$\boxed{\Gamma \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}}$

L-Unit
$$\frac{}{\Gamma \vdash () : \mathbb{1} \rhd ()}$$

L-Lit
$$\frac{}{\Gamma \vdash \mathtt{b} : \mathbb{B} \rhd \mathtt{b}}$$

L-Var
$$\frac{\mathtt{x} : \theta \in \Gamma}{\Gamma \vdash \mathtt{x} : \theta \rhd \mathtt{x}}$$

L-Fun
$$\frac{\mathtt{x} : \theta \rhd \dot{\mathtt{x}} \in \mathcal{L}}{\Gamma \vdash \mathtt{x} : \theta \rhd \dot{\mathtt{x}}}$$

L-Abs
$$\frac{\mathtt{x} : \theta_1, \Gamma \vdash \mathtt{e} : \theta_2 \rhd \dot{\mathtt{e}}}{\Gamma \vdash \lambda \mathtt{x} : \lfloor \theta_1 \rfloor \Rightarrow \mathtt{e} : \theta_1 \to \theta_2 \rhd \lambda \mathtt{x} : \theta_1 \Rightarrow \dot{\mathtt{e}}}$$

L-App
$$\frac{\Gamma \vdash \mathtt{e}_2 : \theta_1 \to \theta_2 \rhd \dot{\mathtt{e}}_2 \qquad \Gamma \vdash \mathtt{e}_1 : \theta_1 \rhd \dot{\mathtt{e}}_1}{\Gamma \vdash \mathtt{e}_2\ \ \mathtt{e}_1 : \theta_2 \rhd \dot{\mathtt{e}}_2\ \ \dot{\mathtt{e}}_1}$$

L-Pair
$$\frac{\Gamma \vdash \mathtt{e}_1 : \theta_1 \rhd \dot{\mathtt{e}}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \theta_2 \rhd \dot{\mathtt{e}}_2}{\Gamma \vdash (\mathtt{e}_1, \mathtt{e}_2) : \theta_1 \times \theta_2 \rhd (\dot{\mathtt{e}}_1, \dot{\mathtt{e}}_2)}$$

L-Proj
$$\frac{\Gamma \vdash \mathtt{e} : \theta_1 \times \theta_2 \rhd \dot{\mathtt{e}}}{\Gamma \vdash \pi_b\ \ \mathtt{e} : \mathtt{ite}(\mathtt{b}, \theta_1, \theta_2) \rhd \pi_b\ \ \dot{\mathtt{e}}}$$

L-Let
$$\frac{\Gamma \vdash \mathtt{e}_1 : \theta_1 \rhd \dot{\mathtt{e}}_1 \qquad \mathtt{x} : \theta_1, \Gamma \vdash \mathtt{e}_2 : \theta_2 \rhd \dot{\mathtt{e}}_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : \theta_2 \rhd \mathtt{let}\ \mathtt{x} = \dot{\mathtt{e}}_1\ \mathtt{in}\ \dot{\mathtt{e}}_2}$$

L-Coerce
$$\frac{\Gamma \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}} \qquad \theta \rightarrowtail \theta' \rhd \uparrow}{\Gamma \vdash \mathtt{e} : \theta' \rhd \uparrow \dot{\mathtt{e}}}$$

L-Ctor$_1$
$$\frac{\mathtt{data}\ \mathtt{T} = \overline{\mathtt{C}\ \eta} \in \Sigma \qquad \Gamma \vdash \mathtt{e} : \eta_i \rhd \dot{\mathtt{e}}}{\Gamma \vdash \mathtt{C}_i\ \ \mathtt{e} : \mathtt{T} \rhd \mathtt{C}_i\ \ \dot{\mathtt{e}}}$$

L-Ctor$_2$
$$\frac{\widehat{\mathtt{C}}_i : \theta_i \to \Psi\widehat{\mathtt{T}} \in \mathcal{S}_I \qquad \Gamma \vdash \mathtt{e} : \theta_i \rhd \dot{\mathtt{e}}}{\Gamma \vdash \mathtt{C}_i\ \ \mathtt{e} : \Psi\widehat{\mathtt{T}} \rhd \widehat{\mathtt{C}}_i\ \ \dot{\mathtt{e}}}$$

L-If$_1$
$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathbb{B} \rhd \dot{\mathtt{e}}_0 \qquad \Gamma \vdash \mathtt{e}_1 : \theta \rhd \dot{\mathtt{e}}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \theta \rhd \dot{\mathtt{e}}_2}{\Gamma \vdash \mathtt{if}\ \ \mathtt{e}_0\ \ \mathtt{then}\ \ \mathtt{e}_1\ \ \mathtt{else}\ \ \mathtt{e}_2 : \theta \rhd \mathtt{if}\ \ \dot{\mathtt{e}}_0\ \ \mathtt{then}\ \ \dot{\mathtt{e}}_1\ \ \mathtt{else}\ \ \dot{\mathtt{e}}_2}$$

L-If$_2$
$$\frac{\Gamma \vdash \mathtt{e}_0 : \widehat{\mathbb{B}} \rhd \dot{\mathtt{e}}_0 \qquad \lightning\theta \rhd \widehat{\mathtt{ite}} \qquad \Gamma \vdash \mathtt{e}_1 : \theta \rhd \dot{\mathtt{e}}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \theta \rhd \dot{\mathtt{e}}_2}{\Gamma \vdash \mathtt{if}\ \ \mathtt{e}_0\ \ \mathtt{then}\ \ \mathtt{e}_1\ \ \mathtt{else}\ \ \mathtt{e}_2 : \theta \rhd \widehat{\mathtt{ite}}\ \ \dot{\mathtt{e}}_0\ \ \dot{\mathtt{e}}_1\ \ \dot{\mathtt{e}}_2}$$

L-Match$_1$
$$\frac{\mathtt{data}\ \mathtt{T} = \overline{\mathtt{C}\ \eta} \in \Sigma \qquad \Gamma \vdash \mathtt{e}_0 : \mathtt{T} \rhd \dot{\mathtt{e}}_0 \qquad \forall i.\ \mathtt{x} : \eta_i, \Gamma \vdash \mathtt{e}_i : \theta' \rhd \dot{\mathtt{e}}_i}{\Gamma \vdash \mathtt{match}\ \ \mathtt{e}_0\ \ \mathtt{with}\ \ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{e}} : \theta' \rhd \mathtt{match}\ \ \dot{\mathtt{e}}_0\ \ \mathtt{with}\ \ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \dot{\mathtt{e}}}}$$

L-Match$_2$
$$\frac{\widehat{\mathtt{match}} : \Psi\widehat{\mathtt{T}} \to \overline{(\theta \to \theta')} \to \theta' \in \mathcal{S}_E \qquad \Gamma \vdash \mathtt{e}_0 : \Psi\widehat{\mathtt{T}} \rhd \dot{\mathtt{e}}_0 \qquad \forall i.\ \mathtt{x} : \theta_i, \Gamma \vdash \mathtt{e}_i : \theta' \rhd \dot{\mathtt{e}}_i}{\Gamma \vdash \mathtt{match}\ \ \mathtt{e}_0\ \ \mathtt{with}\ \ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{e}} : \theta' \rhd \widehat{\mathtt{match}}\ \ \dot{\mathtt{e}}_0\ \ \overline{(\lambda \mathtt{x} : \theta \Rightarrow \dot{\mathtt{e}})}}$$

Figure B.2. Declarative lifting rules

B.2 Algorithmic Lifting

Figure B.3 and Figure B.4 presents the full rules of the lifting algorithm, $\Sigma; \Gamma \vdash \mathtt{e} : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}} \mid \mathcal{C}$.

$$\boxed{\Gamma \vdash \mathtt{e} : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}} \mid \mathcal{C}}$$

A-Unit
$$\frac{}{\Gamma \vdash () : \mathbb{1} \sim \mathtt{X} \rhd () \mid \mathtt{X} = \mathbb{1}}$$

A-Lit
$$\frac{}{\Gamma \vdash \mathtt{b} : \mathbb{B} \sim \mathtt{X} \rhd \mathtt{b} \mid \mathtt{X} = \mathbb{B}}$$

A-Var
$$\frac{\mathtt{x} : \eta \sim \mathtt{X} \in \Gamma}{\Gamma \vdash \mathtt{x} : \eta \sim \mathtt{X}' \rhd \%\uparrow(\mathtt{X},\mathtt{X}';\mathtt{x}) \mid \%\uparrow(\mathtt{X},\mathtt{X}')}$$

A-Fun
$$\frac{\mathtt{fn}\ \mathtt{x}{:}\eta\ =\ \mathtt{e} \in \Sigma}{\Gamma \vdash \mathtt{x} : \eta \sim \mathtt{X} \rhd \%\mathtt{x}(\mathtt{X}) \mid \%\mathtt{x}(\mathtt{X})}$$

A-Abs
$$\frac{\mathtt{X}_1, \mathtt{X}_2 \text{ fresh} \qquad \mathtt{x} : \eta_1 \sim \mathtt{X}_1, \Gamma \vdash \mathtt{e} : \eta_2 \sim \mathtt{X}_2 \rhd \dot{\mathtt{e}} \mid \mathcal{C}}{\Gamma \vdash \lambda \mathtt{x}{:}\eta_1 \Rightarrow \mathtt{e} : \eta_1 \to \eta_2 \sim \mathtt{X} \rhd \lambda \mathtt{x}{:}\mathtt{X}_1 \Rightarrow \dot{\mathtt{e}} \mid \mathtt{X}_1 \in [\eta_1], \mathtt{X}_2 \in [\eta_2], \mathtt{X} = \mathtt{X}_1 \to \mathtt{X}_2, \mathcal{C}}$$

A-App
$$\frac{\mathtt{X}_1 \text{ fresh} \qquad \mathtt{x}_2 : \eta_1 \to \eta_2 \sim \mathtt{X} \in \Gamma \qquad \Gamma \vdash \mathtt{x}_1 : \eta_1 \sim \mathtt{X}_1 \rhd \dot{\mathtt{e}}_1 \mid \mathcal{C}}{\Gamma \vdash \mathtt{x}_2\ \ \mathtt{x}_1 : \eta_2 \sim \mathtt{X}_2 \rhd \mathtt{x}_2\ \ \dot{\mathtt{e}}_1 \mid \mathtt{X}_1 \in [\eta_1], \mathtt{X} = \mathtt{X}_1 \to \mathtt{X}_2, \mathcal{C}}$$

A-Let
$$\frac{\mathtt{X}_1 \text{ fresh} \qquad \Gamma \vdash \mathtt{e}_1 : \eta_1 \sim \mathtt{X}_1 \rhd \dot{\mathtt{e}}_1 \mid \mathcal{C}_1 \qquad \mathtt{x} : \eta_1 \sim \mathtt{X}_1, \Gamma \vdash \mathtt{e}_2 : \eta_2 \sim \mathtt{X}_2 \rhd \dot{\mathtt{e}}_2 \mid \mathcal{C}_2}{\Gamma \vdash \mathtt{let}\ \mathtt{x}{:}\eta_1\ =\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : \eta_2 \sim \mathtt{X}_2 \rhd \mathtt{let}\ \mathtt{x}{:}\mathtt{X}_1\ =\ \dot{\mathtt{e}}_1\ \mathtt{in}\ \dot{\mathtt{e}}_2 \mid \mathtt{X}_1 \in [\eta_1], \mathcal{C}_1, \mathcal{C}_2}$$

A-Pair
$$\frac{\mathtt{x}_1 : \eta_1 \sim \mathtt{X}_1 \in \Gamma \qquad \mathtt{x}_2 : \eta_2 \sim \mathtt{X}_2 \in \Gamma}{\Gamma \vdash (\mathtt{x}_1, \mathtt{x}_2) : \eta_1 \times \eta_2 \sim \mathtt{X} \rhd (\mathtt{x}_1, \mathtt{x}_2) \mid \mathtt{X} = \mathtt{X}_1 \times \mathtt{X}_2}$$

A-Proj
$$\frac{\mathtt{ite}(\mathtt{b},\mathtt{X}_2,\mathtt{X}_1) \text{ fresh} \qquad \mathtt{x} : \eta_1 \times \eta_2 \sim \mathtt{X} \in \Gamma}{\begin{array}{l}\Gamma \vdash \pi_b\ \mathtt{x} : \mathtt{ite}(\mathtt{b},\eta_1,\eta_2) \sim \mathtt{ite}(\mathtt{b},\mathtt{X}_1,\mathtt{X}_2) \rhd \pi_b\ \mathtt{x} \\ \quad \mid \mathtt{ite}(\mathtt{b},\mathtt{X}_2,\mathtt{X}_1) \in [\mathtt{ite}(\mathtt{b},\eta_2,\eta_1)], \mathtt{X} = \mathtt{X}_1 \times \mathtt{X}_2\end{array}}$$

Figure B.3. Algorithmic lifting rules

$\boxed{\Gamma \vdash \mathtt{e} : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}} \mid \mathcal{C}}$

A-If
$$\frac{\mathtt{x}_0 : \mathbb{B} \sim \mathtt{X}_0 \in \Gamma \qquad \Gamma \vdash \mathtt{e}_1 : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}}_1 \mid \mathcal{C}_1 \qquad \Gamma \vdash \mathtt{e}_2 : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}}_2 \mid \mathcal{C}_2}{\Gamma \vdash \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 : \eta \sim \mathtt{X} \rhd \%\mathtt{ite}(\mathtt{X}_0,\mathtt{X};\mathtt{x}_0,\dot{\mathtt{e}}_1,\dot{\mathtt{e}}_2) \mid \%\mathtt{ite}(\mathtt{X}_0,\mathtt{X}),\mathcal{C}_1,\mathcal{C}_2}$$

A-Ctor
$$\frac{\mathtt{data}\ \mathtt{T} = \overline{\mathtt{C}\ \eta} \in \Sigma \qquad \mathtt{X}_i \text{ fresh} \qquad \Gamma \vdash \mathtt{x} : \eta_i \sim \mathtt{X}_i \rhd \dot{\mathtt{e}} \mid \mathcal{C}}{\Gamma \vdash \mathtt{C}_i\ \ \mathtt{x} : \mathtt{T} \sim \mathtt{X} \rhd \%\mathtt{C}_i(\mathtt{X}_i,\mathtt{X};\dot{\mathtt{e}}) \mid \mathtt{X}_i \in [\eta_i], \%\mathtt{C}_i(\mathtt{X}_i,\mathtt{X}), \mathcal{C}}$$

A-Match
$$\frac{\begin{array}{c}\mathtt{data}\ \mathtt{T} = \overline{\mathtt{C}\ \eta} \in \Sigma \\ \overline{\mathtt{X}} \text{ fresh} \qquad \mathtt{x}_0 : \mathtt{T} \sim \mathtt{X}_0 \in \Gamma \qquad \forall i.\ \mathtt{x} : \eta_i \sim \mathtt{X}_i, \Gamma \vdash \mathtt{e}_i : \eta' \sim \mathtt{X}' \rhd \dot{\mathtt{e}}_i \mid \mathcal{C}_i\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{match}\ \mathtt{x}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x} \Rightarrow \mathtt{e}} : \eta' \sim \mathtt{X}' \rhd \%\mathtt{match}(\mathtt{X}_0,\overline{\mathtt{X}},\mathtt{X}';\mathtt{x}_0,\overline{\dot{\mathtt{e}}}) \\ \quad \mid \overline{\mathtt{X} \in [\eta]}, \%\mathtt{match}(\mathtt{X}_0,\overline{\mathtt{X}},\mathtt{X}'), \overline{\mathcal{C}}\end{array}}$$

Figure B.4. Algorithmic lifting rules (cont.)

B.3 Metatheory of Lifting

We say a lifting context $\mathcal{L}$ is well-typed (under Σ) if and only if, for any $\mathtt{x} : \theta \rhd \dot{\mathtt{x}} \in \mathcal{L}$, $\Sigma; \cdot \vdash \mathtt{x} : \lfloor\theta\rfloor$ and $\Sigma; \cdot \vdash \dot{\mathtt{x}} : \theta$. A lifting context is derivable, denoted by $\vdash \mathcal{L}$, if and only if, for any $\mathtt{x} : \theta \rhd \dot{\mathtt{x}} \in \mathcal{L}$, $\mathtt{fn}\ \mathtt{x}{:}\lfloor\theta\rfloor\ =\ \mathtt{e} \in \Sigma$ and $\mathtt{fn}\ \dot{\mathtt{x}}{:}\theta\ =\ \dot{\mathtt{e}} \in \Sigma$ for some $\mathtt{e}$ and $\dot{\mathtt{e}}$, such that $\mathcal{S}; \mathcal{L}; \Sigma; \cdot \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}$. A lifting context is n-valid, denoted by $\vDash_n \mathcal{L}$, if and only if, for any $\mathtt{x} : \theta \rhd \dot{\mathtt{x}} \in \mathcal{L}$, $(\mathtt{x}, \dot{\mathtt{x}}) \in \mathcal{E}_n[\![\theta]\!]$. If $\vDash_n \mathcal{L}$ for any n, we say $\mathcal{L}$ is valid, denoted by $\vDash \mathcal{L}$. Obviously, derivability or validity implies well-typedness.

The lemmas and theorems in this section assume a well-typed global context Σ. We also (explicitly or implicitly) use some standard results about $\lambda_{\mathtt{OADT}\Psi}$ that are proved in the Coq formalization [60]: weakening lemmas, substitution lemmas, preservation theorem and canonical forms of values. We do not state these lemmas formally here, as they are all standard.

Lemma B.3.1. $\lfloor\eta\rfloor = \eta$.

Proof. By routine induction on η. □

Lemma B.3.2 (Regularity of mergeability)**.** $\curlywedge\theta \rhd \widehat{\mathtt{ite}}$ *implies* $\cdot \vdash \widehat{\mathtt{ite}} : \hat{\mathbb{B}} \to \theta \to \theta \to \theta$.

Proof. By routine induction on the derivation of $\uparrow\!\!\!\!\uparrow\theta \rhd \widehat{\texttt{ite}}$ and applying typing rules as needed. □

Lemma B.3.3 (Regularity of coercibility)**.** $\theta \rightarrowtail \theta' \rhd \uparrow$ *implies* $\lfloor\theta\rfloor = \lfloor\theta'\rfloor$ *and* $\cdot \vdash \uparrow : \theta \to \theta'$.

Proof. By routine induction on the derivation of $\theta \rightarrowtail \theta' \rhd \uparrow$ and applying typing rules as needed. □

The following regularity theorem ensures that all lifted expressions are well-typed, which in turn provides the security guarantees, as well-typed programs are oblivious by the obliviousness theorem.

Theorem B.3.4 (Regularity of declarative lifting)**.** *Suppose* $\mathcal{L}$ *is well-typed and* $\mathcal{S}; \mathcal{L}; \Sigma; \Gamma \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}$. *We have* $\Sigma; \lfloor\Gamma\rfloor \vdash \mathtt{e} : \lfloor\theta\rfloor$ *and* $\Sigma; \Gamma \vdash \dot{\mathtt{e}} : \theta$.

Proof. By induction on the derivation of the declarative lifting judgment.

The cases on L-Unit, L-Lit, L-Var and L-Fun are trivial.

The cases on L-Abs, L-App, L-Let, L-Pair, L-Proj, L-If$_1$, L-Ctor$_1$ and L-Match$_1$ are straightforward to prove, as they are simply congruence cases. L-Ctor$_1$ and L-Match$_1$ also rely on Lemma B.3.1. Here we show only the proof on L-Abs. Other cases are similar.

To prove $\Gamma \vdash \lambda \mathtt{x}{:}\theta_1 \Rightarrow \dot{\mathtt{e}} : \theta_1 \to \theta_2$, by T-Abs, we need to show $\mathtt{x} : \theta_1, \Gamma \vdash \dot{\mathtt{e}} : \theta_2$, which follows immediately by the induction hypothesis. The side condition of θ_1 being well-kinded under Γ is immediate from the fact that specification types are well-kinded and the weakening lemma. The other part of this case, $\Gamma \vdash \lambda \mathtt{x}{:}\lfloor\theta_1\rfloor \Rightarrow \mathtt{e} : \lfloor\theta_1 \to \theta_2\rfloor = \lfloor\theta_1\rfloor \to \lfloor\theta_2\rfloor$ proceeds similarly.

Case L-If$_2$: By Lemma B.3.2, we know $\cdot \vdash \widehat{\texttt{ite}} : \widehat{\mathbb{B}} \to \theta \to \theta \to \theta$. Applying T-App and the induction hypothesis, we have $\Gamma \vdash \widehat{\texttt{ite}}\ \dot{\mathtt{e}}_0\ \dot{\mathtt{e}}_1\ \dot{\mathtt{e}}_2 : \theta$, as desired. The other half is straightforward.

Cases L-Ctor$_2$ and L-Match$_2$ rely on the properties of structures $\mathcal{S}_I$ and $\mathcal{S}_E$, but otherwise proceed similarly to other cases.

Case L-Coerce: By the induction hypothesis, we have $\lfloor\Gamma\rfloor \vdash \mathtt{e} : \lfloor\theta\rfloor$ and $\Gamma \vdash \dot{\mathtt{e}} : \theta$. Since $\lfloor\theta\rfloor = \lfloor\theta'\rfloor$ by Lemma B.3.3, $\lfloor\Gamma\rfloor \vdash \mathtt{e} : \lfloor\theta'\rfloor$ as required. On the other hand, $\cdot \vdash \uparrow : \theta \to \theta'$ by Lemma B.3.3. It follows immediately that $\Gamma \vdash \uparrow\dot{\mathtt{e}} : \theta'$. □

Recall that a substitution in the denotation of typing contexts maps names to pairs of related values. In the next lemma (and the rest of this section), we use the notations σ_1 and σ_2 for the substitution projections, i.e., substitutions that only use the first or the second component of the pairs.

Lemma B.3.5 (Multi-substitution)**.** *Let* $\sigma \in \mathcal{G}_n[\![\Gamma]\!]$*. If* $\Gamma \vdash \mathtt{e} : \theta$*, then* $\cdot \vdash \sigma_2(\mathtt{e}) : \theta$*, and if* $\lfloor \Gamma \rfloor \vdash \mathtt{e} : \eta$*, then* $\cdot \vdash \sigma_1(\mathtt{e}) : \eta$*.*

Proof. By routine induction on the structure of Γ, and applying substitution lemma when needed. Note that θ and η do not contain any local variables, so substitution on these types does nothing. □

Lemma B.3.6. *Suppose* $\cdot \vdash \eta :: *^{\mathsf{P}}$*. We have* $(\mathtt{v}, \mathtt{v}) \in \mathcal{V}_n[\![\eta]\!]$ *for any* $\mathtt{v} : \eta$*. Conversely, if* $(\mathtt{v}, \mathtt{v}') \in \mathcal{V}_n[\![\eta]\!]$ *for* $n > 0$*, then* $\mathtt{v} = \mathtt{v}'$*.*

Proof. By routine induction on the structure of η. □

Lemma B.3.7 (Anti-monotonicity)**.** *If* $m \leq n$*, then* $\mathcal{V}_n[\![\theta]\!] \subseteq \mathcal{V}_m[\![\theta]\!]$*, and* $\mathcal{E}_n[\![\theta]\!] \subseteq \mathcal{E}_m[\![\theta]\!]$*, and* $\mathcal{G}_n[\![\Gamma]\!] \subseteq \mathcal{G}_m[\![\Gamma]\!]$*, and* $\vDash_n \mathcal{L} \implies \vDash_m \mathcal{L}$*.*

Proof. To prove the case of value interpretation, proceed by routine induction on θ. The cases of other interpretations are straightforward. □

Lemma B.3.8 (Correctness of mergeability)**.** *Suppose* $\curlywedge\theta \rhd \widehat{\mathtt{ite}}$*. If* $(\mathtt{v}_1, \dot{\mathtt{v}}_1) \in \mathcal{V}_n[\![\theta]\!]$ *and* $(\mathtt{v}_2, \dot{\mathtt{v}}_2) \in \mathcal{V}_n[\![\theta]\!]$*, and* $\widehat{\mathtt{ite}}\ \mathtt{[b]}\ \dot{\mathtt{v}}_1\ \dot{\mathtt{v}}_2 \longrightarrow^* \dot{\mathtt{v}}$*, then* $(\mathtt{ite(b,v_1,v_2)}, \dot{\mathtt{v}}) \in \mathcal{V}_n[\![\theta]\!]$*.*

Proof. By induction on the derivation of mergeability. We assume $n > 0$ because otherwise it is trivial. The cases when θ is $\mathbb{1}$ or $\widehat{\mathbb{B}}$ are trivial. The case of product type is routine, similar to the case of function type.

Case on function type $\theta_1 \to \theta_2$: By assumption,

- $\mathtt{v}_1 = \lambda \mathtt{z} \Rightarrow \mathtt{e}_1$ for some $\mathtt{e}_1$
- $\dot{\mathtt{v}}_1 = \lambda \mathtt{z} \Rightarrow \dot{\mathtt{e}}_1$ for some $\dot{\mathtt{e}}_1$
- $\mathtt{v}_2 = \lambda \mathtt{z} \Rightarrow \mathtt{e}_2$ for some $\mathtt{e}_2$

- $\dot{\mathtt{v}}_2 = \lambda \mathtt{z} \Rightarrow \dot{\mathtt{e}}_2$ for some $\dot{\mathtt{e}}_2$

Suppose $\widehat{\mathtt{ite}}\ \mathtt{[b]}\ \dot{\mathtt{v}}_1\ \dot{\mathtt{v}}_2 \longrightarrow^* \dot{\mathtt{v}}$, i.e., $\dot{\mathtt{v}} = \lambda \mathtt{z} \Rightarrow \widehat{\mathtt{ite}}_2\ \mathtt{[b]}\ (\dot{\mathtt{v}}_1\ \mathtt{z})\ (\dot{\mathtt{v}}_2\ \mathtt{z})$. Suppose $i < n$, and $(\mathtt{u}, \dot{\mathtt{u}}) \in \mathcal{V}_i[\![\theta_1]\!]$. We want to show $(\mathtt{ite(b,[u/z]e_1,[u/z]e_2)}, \widehat{\mathtt{ite}}_2\ \mathtt{[b]}\ (\dot{\mathtt{v}}_1\ \dot{\mathtt{u}})\ (\dot{\mathtt{v}}_2\ \dot{\mathtt{u}})) \in \mathcal{E}_i[\![\theta_2]\!]$. Suppose $j < i$. We have the following trace:

$$\begin{aligned}
\widehat{\mathtt{ite}}_2\ \mathtt{[b]}\ (\dot{\mathtt{v}}_1\ \dot{\mathtt{u}})\ (\dot{\mathtt{v}}_2\ \dot{\mathtt{u}}) &\longrightarrow^1 \widehat{\mathtt{ite}}_2\ \mathtt{[b]}\ (\dot{\mathtt{v}}_1\ \dot{\mathtt{u}})\ (\mathtt{[\dot{u}/z]}\dot{\mathtt{e}}_2) \\
&\longrightarrow^{j_2} \widehat{\mathtt{ite}}_2\ \mathtt{[b]}\ (\dot{\mathtt{v}}_1\ \dot{\mathtt{u}})\ \dot{\mathtt{w}}_2 \\
&\longrightarrow^1 \widehat{\mathtt{ite}}_2\ \mathtt{[b]}\ (\mathtt{[\dot{u}/z]}\dot{\mathtt{e}}_1)\ \dot{\mathtt{w}}_2 \\
&\longrightarrow^{j_1} \widehat{\mathtt{ite}}_2\ \mathtt{[b]}\ \dot{\mathtt{w}}_1\ \dot{\mathtt{w}}_2 \\
&\longrightarrow^{j_3} \dot{\mathtt{w}}
\end{aligned}$$

where $j = j_1 + j_2 + j_3 + 2$, with $\mathtt{[\dot{u}/z]}\dot{\mathtt{e}}_2 \longrightarrow^{j_2} \dot{\mathtt{w}}_2$ and $\mathtt{[\dot{u}/z]}\dot{\mathtt{e}}_1 \longrightarrow^{j_1} \dot{\mathtt{w}}_1$. We instantiate the assumptions with i and $\mathtt{u}$ and $\dot{\mathtt{u}}$ to get:

- $(\mathtt{[u/z]}\mathtt{e}_1, \mathtt{[\dot{u}/z]}\dot{\mathtt{e}}_1) \in \mathcal{E}_i[\![\theta_2]\!]$
- $(\mathtt{[u/z]}\mathtt{e}_2, \mathtt{[\dot{u}/z]}\dot{\mathtt{e}}_2) \in \mathcal{E}_i[\![\theta_2]\!]$

It then follows that:

- $\mathtt{[u/z]}\mathtt{e}_1 \longrightarrow^* \mathtt{w}_1$ for some value $\mathtt{w}_1$
- $(\mathtt{w}_1, \dot{\mathtt{w}}_1) \in \mathcal{V}_{i-j_1}[\![\theta_2]\!]$
- $\mathtt{[u/z]}\mathtt{e}_2 \longrightarrow^* \mathtt{w}_2$ for some value $\mathtt{w}_2$
- $(\mathtt{w}_2, \dot{\mathtt{w}}_2) \in \mathcal{V}_{i-j_2}[\![\theta_2]\!]$

Thus $\mathtt{ite(b,[u/z]e_1,[u/z]e_2)} \longrightarrow^* \mathtt{ite(b,w_1,w_2)}$. It remains to show $(\mathtt{ite(b,w_1,w_2)}, \dot{\mathtt{w}}) \in \mathcal{V}_{i-j}[\![\theta_2]\!]$, but it follows immediately by the induction hypothesis and Lemma B.3.7.

Case on Ψ-type $\Psi\widehat{\mathtt{T}}$: By assumption,

- $\dot{\mathtt{v}}_1 = \langle \mathtt{k}_1, \widehat{\mathtt{v}}_1 \rangle$ for some values $\mathtt{k}_1$ and $\widehat{\mathtt{v}}_1$
- $\mathtt{r}\ \mathtt{k}_1\ \widehat{\mathtt{v}}_1 \longrightarrow^* \mathtt{v}_1$

- $\dot{\mathtt{v}}_2 = \langle \mathtt{k}_2, \widehat{\mathtt{v}}_2 \rangle$ for some values $\mathtt{k}_2$ and $\widehat{\mathtt{v}}_2$
- $\mathtt{r}\ \mathtt{k}_2\ \widehat{\mathtt{v}}_2 \longrightarrow^* \mathtt{v}_2$

Suppose $\widehat{\mathtt{ite}}\ \mathtt{[b]}\ \dot{\mathtt{v}}_1\ \dot{\mathtt{v}}_2 \longrightarrow^* \dot{\mathtt{v}}$. We get the following trace:

$$
\begin{aligned}
\widehat{\mathtt{ite}}\ \mathtt{[b]}\ \dot{\mathtt{v}}_1\ \dot{\mathtt{v}}_2 &\longrightarrow^* \begin{array}{l}\mathtt{let\ k\ =}\ \pi_1\ \dot{\mathtt{v}}_1 \sqcup \pi_1\ \dot{\mathtt{v}}_2\ \mathtt{in} \\ \langle \mathtt{k}, \mathtt{mux}\ \mathtt{[b]}\ (\Uparrow\ (\pi_1\ \dot{\mathtt{v}}_1)\ \mathtt{k}\ (\pi_2\ \dot{\mathtt{v}}_1)) \\ \qquad\qquad (\Uparrow\ (\pi_1\ \dot{\mathtt{v}}_2)\ \mathtt{k}\ (\pi_2\ \dot{\mathtt{v}}_2))\rangle\end{array} \\
&\longrightarrow^* \mathtt{let\ k\ =\ k_1 \sqcup k_2\ in\ \ldots} \\
&\longrightarrow^* \langle \mathtt{k}, \mathtt{mux}\ \mathtt{[b]}\ (\Uparrow\ (\pi_1\ \dot{\mathtt{v}}_1)\ \mathtt{k}\ (\pi_2\ \dot{\mathtt{v}}_1))\ (\ldots)\rangle \\
&\longrightarrow^* \langle \mathtt{k}, \mathtt{mux}\ \mathtt{[b]}\ (\Uparrow\ \mathtt{k}_1\ \mathtt{k}\ \widehat{\mathtt{v}}_1)\ (\ldots)\rangle \\
&\longrightarrow^* \langle \mathtt{k}, \mathtt{mux}\ \mathtt{[b]}\ \widehat{\mathtt{v}}_1'\ (\Uparrow\ (\pi_1\ \dot{\mathtt{v}}_2)\ \mathtt{k}\ (\pi_2\ \dot{\mathtt{v}}_2))\rangle \\
&\longrightarrow^* \langle \mathtt{k}, \mathtt{mux}\ \mathtt{[b]}\ \widehat{\mathtt{v}}_1'\ (\Uparrow\ \mathtt{k}_2\ \mathtt{k}\ \widehat{\mathtt{v}}_2)\rangle \\
&\longrightarrow^* \langle \mathtt{k}, \mathtt{mux}\ \mathtt{[b]}\ \widehat{\mathtt{v}}_1'\ \widehat{\mathtt{v}}_2'\rangle \\
&\longrightarrow^* \langle \mathtt{k}, \mathtt{ite(b,}\widehat{\mathtt{v}}_1', \widehat{\mathtt{v}}_2'\mathtt{)}\rangle = \dot{\mathtt{v}}
\end{aligned}
$$

with $\mathtt{k}_1 \sqcup \mathtt{k}_2 \longrightarrow^* \mathtt{k}$, $\Uparrow\ \mathtt{k}_1\ \mathtt{k}\ \widehat{\mathtt{v}}_1 \longrightarrow^* \widehat{\mathtt{v}}_1'$ and $\Uparrow\ \mathtt{k}_2\ \mathtt{k}\ \widehat{\mathtt{v}}_2 \longrightarrow^* \widehat{\mathtt{v}}_2'$. Because $\mathtt{v}_1 \preccurlyeq \mathtt{k}_1$ by A-O_2, and $\mathtt{k}_1 \sqsubseteq \mathtt{k}$ by A-R_2, we have $\mathtt{v}_1 \preccurlyeq \mathtt{k}$ by A-R_3. It follows that $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}}_1' \longrightarrow^* \mathtt{v}_1$ by A-R_4. Similarly, we get $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}}_2' \longrightarrow^* \mathtt{v}_2$. Hence, $\mathtt{r}\ \mathtt{k}\ \mathtt{ite(b,}\widehat{\mathtt{v}}_1', \widehat{\mathtt{v}}_2'\mathtt{)} \longrightarrow^* \mathtt{ite(b,v_1,v_2)}$. That is to say $(\mathtt{ite(b,v_1,v_2)}, \dot{\mathtt{v}}) \in \mathcal{V}_n[\![\Psi\widehat{\mathtt{T}}]\!]$, as desired. □

Lemma B.3.9 (Correctness of coercibility)**.** *Suppose* $\theta \rightarrowtail \theta' \rhd \uparrow$. *If* $(\mathtt{v}, \dot{\mathtt{v}}) \in \mathcal{V}_n[\![\theta]\!]$ *and* $\uparrow\dot{\mathtt{v}} \longrightarrow^* \dot{\mathtt{v}}'$, *then* $(\mathtt{v}, \dot{\mathtt{v}}') \in \mathcal{V}_n[\![\theta']\!]$.

Proof. By induction on the derivation of coercibility. We only consider $n > 0$ since it is otherwise trivial. The cases on identity coercion and boolean coercion are trivial. The case on $\Psi\widehat{\mathtt{T}} \rightarrowtail \Psi\widehat{\mathtt{T}}'$ is immediate from A-C_1. The case on product type is routine, similar to the case of function type.

Case on $\mathtt{T} \rightarrowtail \Psi\widehat{\mathtt{T}}$: Suppose $(\mathtt{v}, \dot{\mathtt{v}}) \in \mathcal{V}_n[\![\mathtt{T}]\!]$, i.e., $\mathtt{v} = \dot{\mathtt{v}}$ by definition, and $\uparrow\dot{\mathtt{v}} = \uparrow\mathtt{v} \longrightarrow^* \dot{\mathtt{v}}'$. We have the following trace:

$$\begin{aligned}\uparrow\mathtt{v} &\longrightarrow \langle \nu\ \mathtt{v}, \mathtt{s}\ (\nu\ \mathtt{v})\ \mathtt{v}\rangle \\ &\longrightarrow^* \langle \mathtt{k}, \mathtt{s}\ \mathtt{k}\ \mathtt{v}\rangle \\ &\longrightarrow^* \langle \mathtt{k}, \widehat{\mathtt{v}}\rangle = \dot{\mathtt{v}}'\end{aligned}$$

with $\nu\ \mathtt{v} \longrightarrow^* \mathtt{k}$ and $\mathtt{s}\ \mathtt{k}\ \mathtt{v} \longrightarrow^* \widehat{\mathtt{v}}$. Knowing $\mathtt{v} \preccurlyeq \mathtt{k}$ by A-O_3, we have $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{v}$ by A-O_1. But that is $(\mathtt{v}, \langle \mathtt{k}, \widehat{\mathtt{v}}\rangle) \in \mathcal{V}_n[\![\Psi\widehat{\mathtt{T}}]\!]$, as desired.

Case on $\theta_1 \rightarrow \theta_2 \rightarrowtail \theta_1' \rightarrow \theta_2'$: By assumption,

- $\mathtt{v} = \lambda\mathtt{y} \Rightarrow \mathtt{e}$ for some $\mathtt{e}$
- $\dot{\mathtt{v}} = \lambda\mathtt{y} \Rightarrow \dot{\mathtt{e}}$ for some $\dot{\mathtt{e}}$

Suppose $\uparrow\dot{\mathtt{v}} \longrightarrow^* \dot{\mathtt{v}}'$, i.e., $\dot{\mathtt{v}}' = \lambda\mathtt{y} \Rightarrow \uparrow_2(\dot{\mathtt{v}}\ (\uparrow_1\mathtt{y}))$. We want to show $(\mathtt{v}, \dot{\mathtt{v}}') \in \mathcal{V}_n[\![\theta_1' \rightarrow \theta_2']\!]$. Towards this goal fix $i < n$ and $(\mathtt{v}_1, \dot{\mathtt{v}}_1') \in \mathcal{V}_i[\![\theta_1']\!]$. It suffices to show $(\mathtt{[v_1/y]e}, \uparrow_2(\dot{\mathtt{v}}\ (\uparrow_1\dot{\mathtt{v}}_1'))) \in \mathcal{E}_i[\![\theta_2']\!]$. Suppose $j < i$. We have the following trace:

$$\begin{aligned}\uparrow_2(\dot{\mathtt{v}}\ (\uparrow_1\dot{\mathtt{v}}_1')) &\longrightarrow^{j_1} \uparrow_2(\dot{\mathtt{v}}\ \dot{\mathtt{v}}_1) \\ &\longrightarrow^{1} \uparrow_2(\mathtt{[}\dot{\mathtt{v}}_1\mathtt{/y]}\dot{\mathtt{e}}) \\ &\longrightarrow^{j_2} \uparrow_2\dot{\mathtt{v}}_2 \\ &\longrightarrow^{j_3} \dot{\mathtt{v}}_2'\end{aligned}$$

where $j = j_1 + j_2 + j_3 + 1$, with $\uparrow_1\dot{\mathtt{v}}_1' \longrightarrow^{j_1} \dot{\mathtt{v}}_1$ and $\mathtt{[}\dot{\mathtt{v}}_1\mathtt{/y]}\dot{\mathtt{e}} \longrightarrow^{j_2} \dot{\mathtt{v}}_2$. By the induction hypothesis, we get $(\mathtt{v}_1, \dot{\mathtt{v}}_1) \in \mathcal{V}_i[\![\theta_1]\!]$. It follows, by assumption, that $(\mathtt{[v_1/y]e}, \mathtt{[}\dot{\mathtt{v}}_1\mathtt{/y]}\dot{\mathtt{e}}) \in \mathcal{E}_i[\![\theta_2]\!]$. Hence $\mathtt{[v_1/y]e} \longrightarrow^* \mathtt{v}_2$ for some $\mathtt{v}_2$ such that $(\mathtt{v}_2, \dot{\mathtt{v}}_2) \in \mathcal{V}_{i-j_2}[\![\theta_2]\!]$. It then follows by the induction hypothesis that $(\mathtt{v}_2, \dot{\mathtt{v}}_2') \in \mathcal{V}_{i-j_2}[\![\theta_2']\!]$, which concludes the proof by Lemma B.3.7.

□

Theorem B.3.10 (Correctness of declarative lifting of expressions)**.** *Suppose* $\mathcal{S}; \mathcal{L}; \Sigma; \Gamma \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}$ *and* $\vDash_n \mathcal{L}$*. Given a substitution* $\sigma \in \mathcal{G}_n[\![\Gamma]\!]$*, we have* $(\sigma_1(\mathtt{e}), \sigma_2(\dot{\mathtt{e}})) \in \mathcal{E}_n[\![\theta]\!]$*.*

Proof. By induction on the derivation of the declarative lifting judgment. Note that we do not fix step n in the induction. That means we can instantiate it when applying the induction hypothesis, but we also need to discharge $\vDash_n \mathcal{L}$ for the n we pick. We will not explicitly show this side-condition for brevity, because it is simply a consequence of Lemma B.3.7 as long as we pick the same or a smaller step. The well-typedness side-conditions in logical relations are omitted, because they are trivial from Theorem B.3.4 and preservation theorem. In addition, we only consider step $n > 0$ because the cases when $n = 0$ are always trivial.

The cases L-Unit, L-Lit, L-Var and L-Fun are trivial.

Case T-Abs: We need to show $(\lambda \mathtt{x}\mathtt{:}\lfloor\theta\rfloor \Rightarrow \sigma_1(\mathtt{e}), \lambda \mathtt{x}\mathtt{:}\theta \Rightarrow \sigma_2(\dot{\mathtt{e}})) \in \mathcal{E}_n[\![\theta_1 \to \theta_2]\!]$. Suppose $i < n$. Because lambda abstraction can not take step, $i = 0$, and it suffices to show $(\lambda \mathtt{x}\mathtt{:}\lfloor\theta\rfloor \Rightarrow \sigma_1(\mathtt{e}), \lambda \mathtt{x}\mathtt{:}\theta \Rightarrow \sigma_2(\dot{\mathtt{e}})) \in \mathcal{V}_n[\![\theta_1 \to \theta_2]\!]$. To this end fix $i < n$ and suppose that $(\mathtt{v}, \dot{\mathtt{v}}) \in \mathcal{V}_i[\![\theta_1]\!]$ for some $\mathtt{v}$ and $\dot{\mathtt{v}}$. We know $\sigma[\mathtt{x} \mapsto (\mathtt{v}, \dot{\mathtt{v}})] \in \mathcal{G}_i[\![\mathtt{x} : \theta_1, \Gamma]\!]$, by assumption and Lemma B.3.7. Now we specialize the induction hypothesis with $n = i$ and substitution $\sigma[\mathtt{x} \mapsto (\mathtt{v}, \dot{\mathtt{v}})]$ to get $(\sigma_1[\mathtt{x} \mapsto \mathtt{v}](\mathtt{e}), \sigma_2[\mathtt{x} \mapsto \dot{\mathtt{v}}](\dot{\mathtt{e}})) = ([\mathtt{v}/\mathtt{x}]\sigma_1(\mathtt{e}), [\dot{\mathtt{v}}/\mathtt{x}]\sigma_2(\dot{\mathtt{e}})) \in \mathcal{E}_i[\![\theta_2]\!]$, as required.

Case T-App: We need to show $(\sigma_1(\mathtt{e}_2)\ \ \sigma_1(\mathtt{e}_1), \sigma_2(\dot{\mathtt{e}}_2)\ \ \sigma_2(\dot{\mathtt{e}}_1)) \in \mathcal{E}_n[\![\theta_2]\!]$. Suppose that $i < n$, and $\sigma_2(\dot{\mathtt{e}}_2)\ \ \sigma_2(\dot{\mathtt{e}}_1) \longrightarrow^i \dot{\mathtt{v}}$. By the semantics definition, we have the following reduction trace:

$$\begin{aligned}
\sigma_2(\dot{\mathtt{e}}_2)\ \ \sigma_2(\dot{\mathtt{e}}_1) &\longrightarrow^{i_1} \sigma_2(\dot{\mathtt{e}}_2)\ \ \dot{\mathtt{v}}_1 \\
&\longrightarrow^{i_2} \dot{\mathtt{v}}_2\ \ \dot{\mathtt{v}}_1 \\
&\longrightarrow^{1} [\dot{\mathtt{v}}_1/\mathtt{x}]\dot{\mathtt{e}}_2' \\
&\longrightarrow^{i_3} \dot{\mathtt{v}}
\end{aligned}$$

where $\dot{\mathtt{v}}_2$ is $\lambda \mathtt{x} \Rightarrow \dot{\mathtt{e}}_2'$ for some $\dot{\mathtt{e}}_2'$, and $i = i_1 + i_2 + i_3 + 1$, with $\sigma_2(\dot{\mathtt{e}}_1) \longrightarrow^{i_1} \dot{\mathtt{v}}_1$ and $\sigma_2(\dot{\mathtt{e}}_2) \longrightarrow^{i_2} \dot{\mathtt{v}}_2$. Instantiating the two induction hypotheses with step n and σ, we get:

- $(\sigma_1(\mathtt{e}_2), \sigma_2(\dot{\mathtt{e}}_2)) \in \mathcal{E}_n[\![\theta_1 \to \theta_2]\!]$
- $(\sigma_1(\mathtt{e}_1), \sigma_2(\dot{\mathtt{e}}_1)) \in \mathcal{E}_n[\![\theta_1]\!]$

It follows that:

- $\sigma_1(\mathtt{e}_2) \longrightarrow^* \mathtt{v}_2$ for some $\mathtt{v}_2$
- $(\mathtt{v}_2, \dot{\mathtt{v}}_2) \in \mathcal{V}_{n-i_2}[\![\theta_1 \to \theta_2]\!]$
- $\sigma_1(\mathtt{e}_1) \longrightarrow^* \mathtt{v}_1$ for some $\mathtt{v}_1$
- $(\mathtt{v}_1, \dot{\mathtt{v}}_1) \in \mathcal{V}_{n-i_1}[\![\theta_1]\!]$

where $\mathtt{v}_2$ is $\lambda \mathtt{x} \Rightarrow \mathtt{e}_2'$ for some $\mathtt{e}_2'$. It follows that $([\mathtt{v}_1/\mathtt{x}]\,\mathtt{e}_2', [\dot{\mathtt{v}}_1/\mathtt{x}]\,\dot{\mathtt{e}}_2') \in \mathcal{E}_{n-i_1-i_2-1}[\![\theta_2]\!]$, by specializing the value interpretation of function type to $n - i_1 - i_2 - 1$. As $i_3 < n - i_1 - i_2 - 1$, we know $[\mathtt{v}_1/\mathtt{x}]\,\mathtt{e}_2' \longrightarrow^* \mathtt{v}$ for some $\mathtt{v}$, such that $(\mathtt{v}, \dot{\mathtt{v}}) \in \mathcal{V}_{n-i_1-i_2-i_3-1}[\![\theta_2]\!]$. Therefore,

$$\begin{aligned}
\sigma_1(\mathtt{e}_2)\ \ \sigma_1(\mathtt{e}_1) &\longrightarrow^* \sigma_1(\mathtt{e}_2)\ \ \mathtt{v}_1 \\
&\longrightarrow^* \mathtt{v}_2\ \ \mathtt{v}_1 \\
&\longrightarrow [\mathtt{v}_1/\mathtt{x}]\,\mathtt{e}_2' \\
&\longrightarrow^* \mathtt{v}
\end{aligned}$$

and $(\mathtt{v}, \dot{\mathtt{v}}) \in \mathcal{V}_{n-i}[\![\theta_2]\!]$, as desired.

Case L-Let: Observe the trace of the lifted expression:

$$\begin{aligned}
\mathtt{let}\ \mathtt{x} = \sigma_2(\dot{\mathtt{e}}_1)\ \mathtt{in}\ \sigma_2(\dot{\mathtt{e}}_2) &\longrightarrow^{i_1} \mathtt{let}\ \mathtt{x} = \dot{\mathtt{v}}_1\ \mathtt{in}\ \sigma_2(\dot{\mathtt{e}}_2) \\
&\longrightarrow^{1} [\dot{\mathtt{v}}_1/\mathtt{x}]\sigma_2(\dot{\mathtt{e}}_2) \\
&\longrightarrow^{i_2} \dot{\mathtt{v}}
\end{aligned}$$

The rest of the proof is similar to L-App and L-Abs, with the induction hypothesis of the `let`-body specialized to step $n - i_1$.

Case L-Pair: Similarly with the trace:

$$\begin{aligned}
(\sigma_2(\dot{\mathtt{e}}_1), \sigma_2(\dot{\mathtt{e}}_2)) &\longrightarrow^{i_1} (\dot{\mathtt{v}}_1, \sigma_2(\dot{\mathtt{e}}_2)) \\
&\longrightarrow^{i_2} (\dot{\mathtt{v}}_1, \dot{\mathtt{v}}_2)
\end{aligned}$$

Case L-Proj: Similarly with the trace:

$$\begin{aligned}\pi_b\ \sigma_2(\dot{\mathtt{e}}) &\longrightarrow^{i_1} \pi_b\ (\dot{\mathtt{v}}_1,\dot{\mathtt{v}}_2)\\ &\longrightarrow^{1} \mathtt{ite}(\mathtt{b},\dot{\mathtt{v}}_1,\dot{\mathtt{v}}_2)\end{aligned}$$

Case L-If$_1$: Similarly with the trace:

$$\begin{aligned}\mathtt{if}\ \sigma_2(\dot{\mathtt{e}}_0)\ \mathtt{then}\ \sigma_2(\dot{\mathtt{e}}_1)\ \mathtt{else}\ \sigma_2(\dot{\mathtt{e}}_2) &\longrightarrow^{i_0} \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \sigma_2(\dot{\mathtt{e}}_1)\ \mathtt{else}\ \sigma_2(\dot{\mathtt{e}}_2)\\ &\longrightarrow^{1} \mathtt{ite}(\mathtt{b},\sigma_2(\dot{\mathtt{e}}_1),\sigma_2(\dot{\mathtt{e}}_2))\\ &\longrightarrow^{j} \dot{\mathtt{v}}\end{aligned}$$

Case L-Ctor$_1$: Similarly with the trace $\mathtt{C}_i\ \sigma_2(\dot{\mathtt{e}}) \longrightarrow^{i} \mathtt{C}_i\ \dot{\mathtt{v}}$. We also need to apply Lemma B.3.6 to complete the proof.

Case L-Match$_1$: Similarly with the trace:

$$\begin{aligned}\mathtt{match}\ \sigma_2(\dot{\mathtt{e}}_0)\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x}\Rightarrow\sigma_2(\dot{\mathtt{e}})} &\longrightarrow^{j_0} \mathtt{match}\ \mathtt{C}_i\ \dot{\mathtt{v}}_0\ \mathtt{with}\ \overline{\mathtt{C}\ \mathtt{x}\Rightarrow\sigma_2(\dot{\mathtt{e}})}\\ &\longrightarrow^{1} [\dot{\mathtt{v}}_0/\mathtt{x}]\sigma_2(\dot{\mathtt{e}}_i)\\ &\longrightarrow^{j_1} \dot{\mathtt{v}}\end{aligned}$$

Similar to L-Ctor$_1$, Lemma B.3.6 is used.

Case L-If$_2$: Suppose $i < n$. We have the following trace:

$$\begin{aligned}\widehat{\mathtt{ite}}\ \sigma_2(\dot{\mathtt{e}}_0)\ \sigma_2(\dot{\mathtt{e}}_1)\ \sigma_2(\dot{\mathtt{e}}_2) &\longrightarrow^{i_2} \widehat{\mathtt{ite}}\ \sigma_2(\dot{\mathtt{e}}_0)\ \sigma_2(\dot{\mathtt{e}}_1)\ \dot{\mathtt{v}}_2\\ &\longrightarrow^{i_1} \widehat{\mathtt{ite}}\ \sigma_2(\dot{\mathtt{e}}_0)\ \dot{\mathtt{v}}_1\ \dot{\mathtt{v}}_2\\ &\longrightarrow^{i_0} \widehat{\mathtt{ite}}\ [\mathtt{b}]\ \dot{\mathtt{v}}_1\ \dot{\mathtt{v}}_2\\ &\longrightarrow^{i_3} \dot{\mathtt{v}}\end{aligned}$$

where $i = i_0 + i_1 + i_2 + i_3$, with $\sigma_2(\dot{\mathtt{e}}_0) \longrightarrow^{i_0} [\mathtt{b}]$, $\sigma_2(\dot{\mathtt{e}}_1) \longrightarrow^{i_1} \dot{\mathtt{v}}_1$ and $\sigma_2(\dot{\mathtt{e}}_2) \longrightarrow^{i_2} \dot{\mathtt{v}}_2$. By the induction hypothesis, it follows that:

- $\sigma_1(\mathtt{e}_0) \longrightarrow^{*} b$

- $\sigma_1(\mathtt{e}_1) \longrightarrow^* \mathtt{v}_1$ for some $\mathtt{v}_1$
- $(\mathtt{v}_1, \dot{\mathtt{v}}_1) \in \mathcal{V}_{n-i_1}[\![\theta]\!]$
- $\sigma_1(\mathtt{e}_2) \longrightarrow^* \mathtt{v}_2$ for some $\mathtt{v}_2$
- $(\mathtt{v}_2, \dot{\mathtt{v}}_2) \in \mathcal{V}_{n-i_2}[\![\theta]\!]$

Hence we have:

$$\begin{aligned}\mathtt{if}\ \sigma_1(\mathtt{e}_0)\ \mathtt{then}\ \sigma_1(\mathtt{e}_1)\ \mathtt{else}\ \sigma_1(\mathtt{e}_2) &\longrightarrow^* \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \sigma_1(\mathtt{e}_1)\ \mathtt{else}\ \sigma_1(\mathtt{e}_2) \\ &\longrightarrow \mathtt{ite}(\mathtt{b}, \sigma_1(\mathtt{e}_1), \sigma_1(\mathtt{e}_2)) \\ &\longrightarrow^* \mathtt{ite}(\mathtt{b}, \mathtt{v}_1, \mathtt{v}_2)\end{aligned}$$

and $(\mathtt{ite}(\mathtt{b}, \mathtt{v}_1, \mathtt{v}_2), \dot{\mathtt{v}}) \in \mathcal{V}_{n-i}[\![\theta]\!]$ by Lemma B.3.8, as required.

Case L-Ctor$_2$: Similarly with the trace:

$$\begin{aligned}\widehat{\mathtt{C}}_i\ \ \sigma_2(\dot{\mathtt{e}}) &\longrightarrow^{j_1} \widehat{\mathtt{C}}_i\ \ \dot{\mathtt{v}}_i \\ &\longrightarrow^{j_2} \langle \mathtt{k}, \widehat{\mathtt{v}} \rangle = \dot{\mathtt{v}}\end{aligned}$$

We complete the proof by A-I$_1$.

Case L-Match$_2$: Similarly with the trace:

$$\begin{aligned}\widehat{\mathtt{match}}\ \ \sigma_2(\dot{\mathtt{e}}_0)\ \ \overline{(\lambda \mathtt{x} \Rightarrow \sigma_2(\dot{\mathtt{e}}))} &\longrightarrow^{j_0} \widehat{\mathtt{match}}\ \ \dot{\mathtt{v}}_0\ \ \overline{(\lambda \mathtt{x} \Rightarrow \sigma_2(\dot{\mathtt{e}}))} \\ &\longrightarrow^{j_1} \dot{\mathtt{v}}\end{aligned}$$

We complete the proof by A-E$_1$, whose assumptions are discharged by the induction hypothesis.

Case L-Coerce: The goal is $(\sigma_1(\mathtt{e}), \uparrow\sigma_2(\dot{\mathtt{e}})) \in \mathcal{E}_n[\![\theta']\!]$. Suppose $i < n$. We have the trace:

$$\begin{aligned}\uparrow\sigma_2(\dot{\mathtt{e}}) &\longrightarrow^{i_1} \uparrow\dot{\mathtt{v}} \\ &\longrightarrow^{i_2} \dot{\mathtt{v}}'\end{aligned}$$

where $i = i_1 + i_2$, with $\sigma_2(\dot{\mathtt{e}}) \longrightarrow^{i_1} \dot{\mathtt{v}}$. It follows by the induction hypothesis that $\sigma_1(\mathtt{e}) \longrightarrow^* \mathtt{v}$ for some $\mathtt{v}$ such that $(\mathtt{v}, \dot{\mathtt{v}}) \in \mathcal{V}_{n-i_1}[\![\theta]\!]$. Then, by Lemma B.3.9, $(\mathtt{v}, \dot{\mathtt{v}}') \in \mathcal{V}_{n-i}[\![\theta']\!]$, as desired. □

Corollary B.3.11 (Correctness of declarative lifting of closed terms)**.** *Suppose* $\mathcal{S}; \mathcal{L}; \Sigma; \cdot \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}$ *and* $\vDash_n \mathcal{L}$*. We have* $(\mathtt{e}, \dot{\mathtt{e}}) \in \mathcal{E}_n[\![\theta]\!]$*.*

Theorem B.3.12 (Correctness of declarative lifting)**.** $\vdash \mathcal{L}$ *implies* $\vDash \mathcal{L}$*.*

Proof. We need to show $\vDash_n \mathcal{L}$ for any n, i.e., $(\mathtt{x}, \dot{\mathtt{x}}) \in \mathcal{E}_n[\![\theta]\!]$ for any $\mathtt{x} : \theta \rhd \dot{\mathtt{x}}$. The proof is a straightforward induction on n.

The base case is trivial, as $\mathcal{E}_0[\![\theta]\!]$ is a total relation as long as $\mathtt{x}$ and $\dot{\mathtt{x}}$ are well-typed, which is immediate from $\vdash \mathcal{L}$.

Suppose $\vDash_n \mathcal{L}$. We need to show $(\mathtt{x}, \dot{\mathtt{x}}) \in \mathcal{E}_{n+1}[\![\theta]\!]$. From $\vdash \mathcal{L}$, we know $\mathtt{x}$ and $\dot{\mathtt{x}}$ are defined by $\mathtt{e}$ and $\dot{\mathtt{e}}$ in Σ, respectively. It is easy to see it suffices to prove $(\mathtt{e}, \dot{\mathtt{e}}) \in \mathcal{E}_n[\![\theta]\!]$, because $\mathtt{x}$ and $\dot{\mathtt{x}}$ take exactly one step to $\mathtt{e}$ and $\dot{\mathtt{e}}$. But that is immediate by Corollary B.3.11. □

In the following theorem, we write $\sigma(\Gamma_2)$ to obtain the second projection of the typing context with the type variables substituted, i.e., each $\mathtt{x} : \eta \sim \mathtt{X}$ in Γ is mapped to $\mathtt{x} : \sigma(\mathtt{X})$. It may be counterintuitive that we do not require θ to be compatible with η or each pair in the typing context to be compatible, but these side conditions are indeed not needed; if the generated constraints (under empty context) are satisfiable, the compatibility conditions should hold by construction.

Theorem B.3.13 (Soundness of algorithmic lifting of open terms)**.** *Suppose* $\Sigma; \Gamma \vdash \mathtt{e} : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}} \mid \mathcal{C}$*. Given a specification type* θ*, if* $\mathcal{S}; \mathcal{L}; \Sigma; \sigma \vDash \mathtt{X} = \theta, \mathcal{C}$*, then* $\sigma(\dot{\mathtt{e}})$ *elaborates to an expression* $\dot{\mathtt{e}}'$*, such that* $\mathcal{S}; \mathcal{L}; \Sigma; \sigma(\Gamma_2) \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}'$*.*

Proof. By induction on the derivation of the algorithmic lifting judgment. The cases of A-Unit, A-Lit and A-Fun are trivial. The cases of A-Let, A-Pair and A-Proj are similar to A-Abs and A-App, so we only show the proofs of these two cases. The cases of A-Ctor and A-Match are similar to A-If (and other cases), so we only show the proof of A-If.

Case A-Var: Suppose $\sigma \vDash \mathtt{X}' = \theta, \%\uparrow(\mathtt{X}, \mathtt{X}')$. It follows that:

- $\theta = \sigma(\mathtt{X}')$
- $\sigma(\mathtt{X}) \rightarrowtail \sigma(\mathtt{X}') \rhd \uparrow$
- $\%\uparrow(\sigma(\mathtt{X}),\sigma(\mathtt{X}');\mathtt{x}) \rhd \uparrow\mathtt{x}$

Therefore, $\sigma(\%\uparrow(\mathtt{X},\mathtt{X}';\mathtt{x})) = \%\uparrow(\sigma(\mathtt{X}),\sigma(\mathtt{X}');\mathtt{x})$ elaborates to $\uparrow\mathtt{x}$. Since $\mathtt{x} : \eta \sim \mathtt{X} \in \Gamma$ by assumption, $\mathtt{x} : \sigma(\mathtt{X}) \in \sigma(\Gamma_2)$. We can then apply L-Coerce and L-Var to derive $\sigma(\Gamma_2) \vdash \mathtt{x} : \sigma(\mathtt{X}') \rhd \uparrow\mathtt{x}$.

Case A-Abs: Suppose $\sigma \vDash \mathtt{X} = \theta, \mathtt{X}_1 \in [\eta_1], \mathtt{X}_2 \in [\eta_2], \mathtt{X} = \mathtt{X}_1 \to \mathtt{X}_2, \mathcal{C}$. It follows that:

- $\theta = \sigma(\mathtt{X}) = \sigma(\mathtt{X}_1) \to \sigma(\mathtt{X}_2)$
- $\lfloor \sigma(\mathtt{X}_1) \rfloor = \eta_1$
- $\lfloor \sigma(\mathtt{X}_2) \rfloor = \eta_2$

Because $\sigma \vDash \mathtt{X}_2 = \sigma(\mathtt{X}_2), \mathcal{C}$, we have by the induction hypothesis:

- $\sigma(\dot{\mathtt{e}})$ elaborates to some $\dot{\mathtt{e}}'$
- $\mathtt{x} : \sigma(\mathtt{X}_1), \sigma(\Gamma_2) \vdash \mathtt{e} : \sigma(\mathtt{X}_2) \rhd \dot{\mathtt{e}}'$

Thus $\sigma(\lambda \mathtt{x}{:}\mathtt{X}_1 \Rightarrow \dot{\mathtt{e}}) = \lambda \mathtt{x}{:}\sigma(\mathtt{X}_1) \Rightarrow \sigma(\dot{\mathtt{e}})$ elaborates to $\lambda \mathtt{x}{:}\sigma(\mathtt{X}_1) \Rightarrow \dot{\mathtt{e}}'$, and $\sigma(\Gamma_2) \vdash \lambda \mathtt{x}{:}\eta_1 \Rightarrow \mathtt{e} : \sigma(\mathtt{X}_1) \to \sigma(\mathtt{X}_2) \rhd \lambda \mathtt{x}{:}\sigma(\mathtt{X}_1) \Rightarrow \dot{\mathtt{e}}'$, by L-Abs.

Case A-App: Suppose $\sigma \vDash \mathtt{X}_2 = \theta, \mathtt{X}_1 \in [\eta_1], \mathtt{X} = \mathtt{X}_1 \to \mathtt{X}_2, \mathcal{C}$. It follows that:

- $\theta = \sigma(\mathtt{X}_2)$
- $\lfloor \sigma(\mathtt{X}_1) \rfloor = \eta_1$
- $\sigma(\mathtt{X}) = \sigma(\mathtt{X}_1) \to \sigma(\mathtt{X}_2)$

Because $\sigma \vDash \mathtt{X}_1 = \sigma(\mathtt{X}_1), \mathcal{C}$, we have by the induction hypothesis:

- $\sigma(\dot{\mathtt{e}}_1)$ elaborates to some $\dot{\mathtt{e}}_1'$
- $\sigma(\Gamma_2) \vdash \mathtt{x}_1 : \sigma(\mathtt{X}_1) \rhd \dot{\mathtt{e}}_1'$

Therefore, $\sigma(\mathtt{x}_2\ \dot{\mathtt{e}}_1) = \mathtt{x}_2\ \sigma(\dot{\mathtt{e}}_1)$ elaborates to $\mathtt{x}_2\ \dot{\mathtt{e}}'_1$, and $\sigma(\Gamma_2) \vdash \mathtt{x}_2\ \mathtt{x}_1 : \sigma(\mathtt{X}_2) \rhd \mathtt{x}_2\ \dot{\mathtt{e}}'_1$, by L-App and L-Var.

Case A-If: Suppose $\sigma \vDash \mathtt{X} = \theta, \mathtt{\%ite}(\mathtt{X}_0, \mathtt{X}), \mathcal{C}_1, \mathcal{C}_2$. We know $\sigma(\mathtt{X}) = \theta$. Because $\sigma \vDash \mathtt{X} = \sigma(\mathtt{X}), \mathcal{C}_1$ and $\sigma \vDash \mathtt{X} = \sigma(\mathtt{X}), \mathcal{C}_2$, we have by the induction hypothesis:

- $\sigma(\dot{\mathtt{e}}_1)$ elaborates to some $\dot{\mathtt{e}}'_1$
- $\sigma(\Gamma_2) \vdash \mathtt{e}_1 : \sigma(\mathtt{X}) \rhd \dot{\mathtt{e}}'_1$
- $\sigma(\dot{\mathtt{e}}_2)$ elaborates to some $\dot{\mathtt{e}}'_2$
- $\sigma(\Gamma_2) \vdash \mathtt{e}_2 : \sigma(\mathtt{X}) \rhd \dot{\mathtt{e}}'_2$

Since $\sigma \vDash \mathtt{\%ite}(\mathtt{X}_0, \mathtt{X})$, $\mathtt{\%ite}(\sigma(\mathtt{X}_0), \sigma(\mathtt{X}); \mathtt{x}_0, \dot{\mathtt{e}}'_1, \dot{\mathtt{e}}'_2) \rhd \dot{\mathtt{e}}$ for some $\dot{\mathtt{e}}$. By inverting this judgment, we consider two cases. In the first case, the condition has type $\mathbb{B}$:

- $\mathtt{\%ite}(\sigma(\mathtt{X}_0), \sigma(\mathtt{X}); \mathtt{x}_0, \dot{\mathtt{e}}'_1, \dot{\mathtt{e}}'_2) \rhd \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \dot{\mathtt{e}}'_1\ \mathtt{else}\ \dot{\mathtt{e}}'_2$
- $\sigma(\mathtt{X}_0) = \mathbb{B}$

In this case, $\mathtt{\%ite}(\sigma(\mathtt{X}_0), \sigma(\mathtt{X}); \mathtt{x}_0, \sigma(\dot{\mathtt{e}}_1), \sigma(\dot{\mathtt{e}}_2))$ elaborates to $\mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \dot{\mathtt{e}}'_1\ \mathtt{else}\ \dot{\mathtt{e}}'_2$, and, by L-If$_1$ and L-Var, $\sigma(\Gamma_2) \vdash \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 : \sigma(\mathtt{X}) \rhd \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \dot{\mathtt{e}}'_1\ \mathtt{else}\ \dot{\mathtt{e}}'_2$. The second case has condition type $\widehat{\mathbb{B}}$:

- $\lightning\sigma(\mathtt{X}) \rhd \widehat{\mathtt{ite}}$
- $\mathtt{\%ite}(\sigma(\mathtt{X}_0), \sigma(\mathtt{X}); \mathtt{x}_0, \dot{\mathtt{e}}'_1, \dot{\mathtt{e}}'_2) \rhd \widehat{\mathtt{ite}}\ \mathtt{x}_0\ \dot{\mathtt{e}}'_1\ \dot{\mathtt{e}}'_2$
- $\sigma(\mathtt{X}_0) = \widehat{\mathbb{B}}$

In this case, $\mathtt{\%ite}(\sigma(\mathtt{X}_0), \sigma(\mathtt{X}); \mathtt{x}_0, \sigma(\dot{\mathtt{e}}_1), \sigma(\dot{\mathtt{e}}_2))$ elaborates to $\widehat{\mathtt{ite}}\ \mathtt{x}_0\ \dot{\mathtt{e}}'_1\ \dot{\mathtt{e}}'_2$, and, by L-If$_2$ and L-Var, $\sigma(\Gamma_2) \vdash \mathtt{if}\ \mathtt{x}_0\ \mathtt{then}\ \mathtt{e}_1\ \mathtt{else}\ \mathtt{e}_2 : \sigma(\mathtt{X}) \rhd \widehat{\mathtt{ite}}\ \mathtt{x}_0\ \dot{\mathtt{e}}'_1\ \dot{\mathtt{e}}'_2$. □

Corollary B.3.14 (Soundness of algorithmic lifting)**.** *Suppose* $\Sigma; \cdot \vdash \mathtt{e} : \eta \sim \mathtt{X} \rhd \dot{\mathtt{e}} \mid \mathcal{C}$. *Given a specification type* θ*, if* $\mathcal{S}; \mathcal{L}; \Sigma; \sigma \vDash \mathtt{X} = \theta, \mathcal{C}$*, then* $\sigma(\dot{\mathtt{e}})$ *elaborates to an expression* $\dot{\mathtt{e}}'$*, such that* $\mathcal{S}; \mathcal{L}; \Sigma; \cdot \vdash \mathtt{e} : \theta \rhd \dot{\mathtt{e}}'$.